Louisiana State University

LSU Scholarly Repository

Honors Theses

Ogden Honors College

4-2007

Designing a Simulated Testing System for Embedded Systems Development

Abhishek Debchoudry

Follow this and additional works at: https://repository.lsu.edu/honors_etd

Part of the Computer Sciences Commons

Recommended Citation

Debchoudry, Abhishek, "Designing a Simulated Testing System for Embedded Systems Development" (2007). *Honors Theses*. 18.

https://repository.lsu.edu/honors_etd/18

This Thesis is brought to you for free and open access by the Ogden Honors College at LSU Scholarly Repository. It has been accepted for inclusion in Honors Theses by an authorized administrator of LSU Scholarly Repository. For more information, please contact ir@lsu.edu.

Designing a Simulated Testing System for Embedded Systems Development

by

Abhishek Debchoudhury

Undergraduate honors thesis under the direction of

Dr. Gerald Baumgartner

Department of Computer Science

Submitted to the LSU Honors College in partial fulfillment of

the Upper Division Honors Program.

April, 2007

Louisiana State University

& Agricultural and Mechanical College

Baton Rouge, Louisiana

1. ABSTRACT

A critical component of any development process is a viable and comprehensive testing program. A good testing program determines whether the product meets its requirements and finds flaws in the system. While the software development process has seen massive improvements in testing processes and tools, the area of embedded systems testing has not experienced any similar advancement.

Presently, embedded systems are still most often tested by using complete hardware. This hardware is sometimes destroyed in the testing process (destructive testing) thereby making the entire process extremely expensive and slow. To counter this problem, we propose a hardware-in-the-loop (HIL) testing system. We create a virtual test bed which simulates the "real" hardware by utilizing an actual processor and producing output using a host system. This allows the user to test real code using an actual embedded processor without having to risk destroying expensive hardware in the process.

We first describe current hardware testing methodologies and discuss the flaws in the system. We then propose our HIL system and describe how it solves the problems inherent in the current testing system. Following that, we give an overview of a generic HIL system and describe its various parts and functions. We also discuss when and why a HIL system should be used, and propose some guidelines in designing a viable HIL system. Using these guidelines, we describe the implementation of our system. We break down the description into hardware, software and interfaces and describe each part. We discuss why certain choices were made and their effect on the entire system. Special emphasis is put on the software part of the

implementation since this is where the primary contribution of this thesis lies. Finally, we talk about what the current limitations of the system are and where it can be improved. We discuss specific ways the system can be improved in the future to not only make it more abstract but also more cost-effective and user-friendly.

2. TABLE OF CONTENTS

Page

1. Abstract 1

2. Table of Contents 3

3. List of Figures 4

4. Introduction 5

5. Related Work 10

6. Hardware-in-Loop System Design 12

    a. Hardware, Software, and Interface 12

    b. Considerations in System Design 14

7. System Design and Implementation 17

    a. System Setup 17

    b. Hardware 20

    c. Interface 21

    d. Software 23

8. Evaluation 33

9. Conclusions 34

    a. Future Work 35

10. Bibliography 40

3. LIST OF FIGURES

Figure		Page
1.	Number of Processors	5
2.	Hardware-in-Loop System Diagram	12
3.	Picture of the System	19
4.	Block Diagram for the TMS320LF2407	20
5.	User Interface	25
6.	Test Mode User Interface	26
7.	Host System Testing Code part 1	28
8.	Host System Testing Code part 2	29
9.	Example User Code Part 1	30
10.	Example User Code Part 2	31

4. INTRODUCTION

Computer processors are all around us. An average person has contact with some form of a computer chip on a daily basis. Everything from desktop computers and televisions to cell phones and some shoes have computer chips. As Figure 1 shows, the number of embedded computers has increased by more than a factor of five in the last few years. These small embedded processors control almost every aspect of the machines that they are in. Thus, extreme care must be taken to ensure that they work efficiently and correctly at all times during the life time of the device.

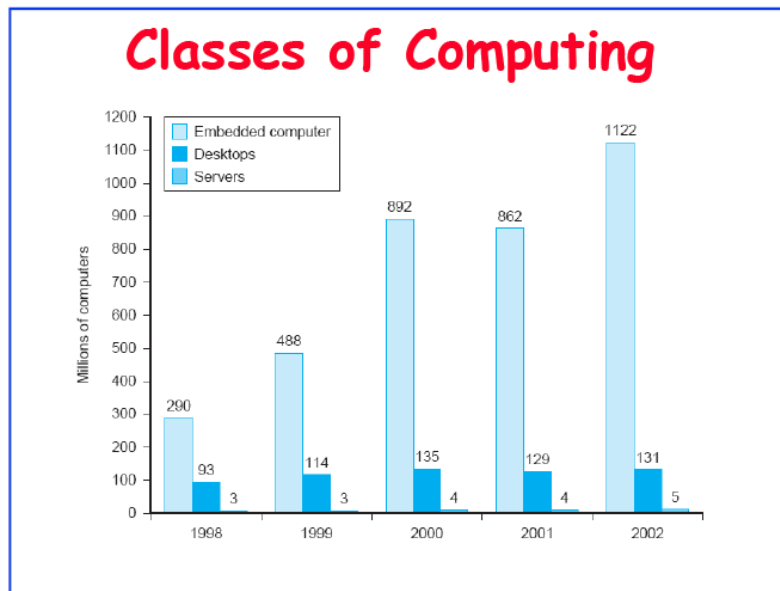

Figure 1: Number of Processors [18]

Embedded devices are generally very small and specialized pieces of equipment. They consist of small quantities of hardware with specialized software written to control their actions. They interact with their environment through the use of sensors and actuators. They

are also usually quite simple. Some embedded computers have operating systems. However, most are so specialized that an operating system adds unnecessary overhead. An average device might contain dozens of these small processors; each controlling a different part of the device.

As the number of these devices has grown, so too has their sophistication. While previously embedded processors controlled very menial tasks, they are now responsible for very complicated work. A modern cell phone is a prime example of this development. Fifteen years ago most phones only had the option to call other phones, store a few phone numbers, and check for new voice mails. However, cell phones on the market now are significantly more complicated. Not only do they perform the tasks mentioned previously, but they also have options to send text messages, take pictures, browse the web, and play video games. Future phones intend to take this even further and allow functions such as touch screens [1]. Thus, as time has passed, more and more code has been written to control each embedded processor.

The problem with this phenomenon is that while the processes used to build the hardware have improved considerably, the processes used to create and test the software have not. The process of manufacturing chips has seen plenty of research and development and thus the hardware can be produced at a relatively high speed and low cost. These manufacturing processes are well quality controlled and are very efficient. However, the software production and testing methods have not seen any such dramatic improvements.

Testing is one of the most complex and time consuming parts of real-time software development [10]. Today, software testing of embedded systems is still done by hand. While

programmers developing applications for desktops have access to various automated testing and debugging tools, their counterparts dealing with embedded systems have few such tools. The existence of this phenomenon is backed up by Grindal et al [12]. This lack of tools in turn has led to some very serious problems with the development of the software:

- The testing process takes a long time: The lack of automated testing tools means that each separate test must be done manually and therefore takes a long time. This slow testing process means that the entire development process is slowed since each feature must be tested before new features can be added in.
- The testing process is expensive: The lack of the software tools for testing means that the testers must use the actual hardware to test their software. Erroneous software can often result in the hardware being tested to be completely destroyed. This makes the testing process and the overall development process very expensive.
- The quality of the testing is poor: Testers must perform a small set of tasks repeatedly for differing inputs. This often causes problems since the testers are human and therefore are prone to make mistakes. Even very minute mistakes can lead to serious issues.

To counter these problems, testers need tools that will not only make their work easier but also quicker, cheaper and more reliable. To solve these needs, we propose a hardware-in-loop (HIL) system that will allow testers to write test code that can then be downloaded onto an embedded device that simulates the control signals to be tested. The tester can then input

what the expected output result is. Once the testing software is started, it will allow the testers to dynamically input test data. The software will then simulate the test code on the hardware and automatically check the results against the expected results. We provide a proof-of-concept design later in this paper detailing a testing system that can be used to test the operation of a traffic light. We not only provide a complete working example of the traffic light system but also the general framework to expand it for other uses.

The benefits of such as system are numerous. By simulating the hardware instead of using the real thing, we drastically reduce the cost of the testing process. For example, instead of actually using a traffic light, testers can simulate the traffic light on our system, thereby reducing the cost of the testing. By having the software dynamically check the actual output against the expected output, we reduce the possibility of human error. While we acknowledge that errors could occur during the actual input of the expected output, the overall possibility of errors is nonetheless drastically reduced with such a system. Using an HIL system instead of a pure simulation system also allows us to test the software on the "real" hardware and get the results back in almost real time [3]. Lastly, the use of testing tools makes the testing process much faster. By providing a framework for testing various systems, we reduce the amount of time it takes to set up a testing system. Instead of having to create a testing system from the ground up, testers can concentrate on tuning the system to meet their specific needs.

The contribution of this thesis to the design is in the implementation of the entire testing framework onto the system. Previously, the system had no way of testing the output against the expected output. We provide the facilities to do such testing. We also implement

the system to allow the tester to bypass the embedded system completely. We implement the code necessary to allow the tester to choose to simulate the entire system on the host computer. We also implement a framework which can be used by the tester to write their test code. We provide all the code necessary for the user to simply drop their own code into the existing framework, input the expected results and run the system. Finally, we have contributed to the general reorganization and optimization of the code. This includes adding in various error-handling routines and fixing bugs in the previous code.

5. RELATED WORK

Scientists and researchers have long acknowledged that work needs to be done in the field of software driven hardware testing. However, unlike their desktop counterparts, embedded system developers have seen few full hardware-in- loop systems capable of testing such a system.

The US military has been researching and working with hardware-in-loop systems. In 1993, a concept of such a system was described by Johnson and Crocker [11]. They gave an overview of how such a system would work. They specified some of the challenges associated with the system both from a technical point of view as well as for a human resource management point of view.

Research is also currently being carried out in academia on this subject. Keyhani et al. have presented a conceptual framework for a HIL model that can be substituted for laboratory experiments [8]. They provide the basic description of a system that uses object oriented coding along with a hardware motor to simulate various hardware. They provide this as a prototype for future use in teaching electrical engineering majors and other students.

While the work on full hardware-in-loop systems has progressed, other researchers have developed full simulation only systems for testing hardware tools. This work has been helped by the personal computer revolution of the 1990s. Nowadays, every engineer has access to a computer and thus can easily write specialized software testing tools. To help with this, private companies have created software that can help. MATLAB is a popular example of

this. They have provided tools to develop and test algorithms [5]. Many engineers use this to test their software before putting it into the hardware. One area where simulation is extensively used is in the development and testing of processor cores. In this field, the core is often split up and tested separately by software in a form of self-testing [14].

Another piece of software that is starting to be used is UPPAAL TRON. This software allows users to test their real time systems online. It allows for users to access the system online and test their systems in a black box environment [2]. Its effectiveness has been verified by other researchers [15].

However, pure software simulation does have its limitations. Since no hardware is involved, algorithms must be written to simulate the hardware. This can cause problems with reliability. Also, pure software systems are not real-time and therefore cannot properly account for systems where timing is pivotal. Lastly, pure simulation software must plot out results while a hardware-in-loop system can use actual digital signals that more closely replicate the hardware's usage [3].

The most important goal of a virtual test bed is to provide a low-cost simulation environment which uses as little hardware as possible while still being reliable [13]. To meet this criteria we describe our system below. It uses both hardware and software thereby negating the disadvantages of a pure simulation system. It also is very cheap due to various hardware and software considerations we make. We describe the advantages as well as disadvantages of these choices later on. We show all this though a description of our proof-of-concept design.

6. HARDWARE-IN-LOOP SYSTEM DESIGN

A hardware-in-loop system (HIL) consists of two main parts: a software input part and a hardware part that simulates the actual hardware [16]. In between these two parts are a series of interfaces that allow the two parts to communicate with each other. They have been historically used for the development of costly system such as satellites and nuclear control reactors [17]. Figure 2 shows a generic HIL system.

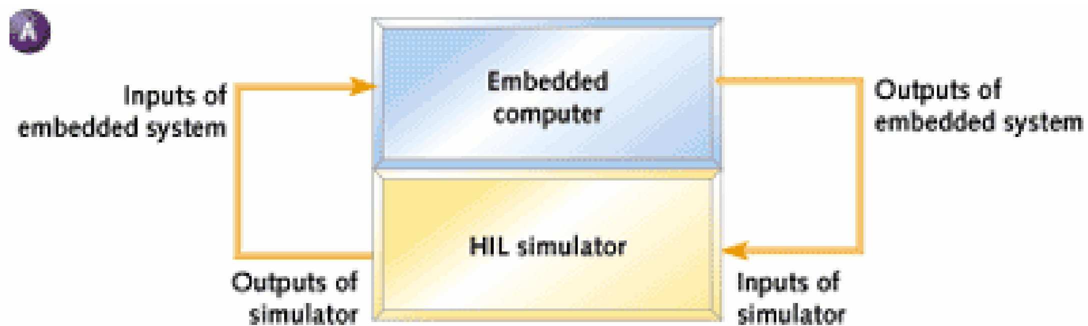

Figure 2: Hardware-in-Loop System Diagram [3]

6.a Hardware, Software, and Interface

The software part is responsible for taking in the inputs to the hardware and sending them to the hardware. This part often has functionality built in to allow for the testers to add in their own code. As described later, our system has a framework setup allowing users to put in their code with relative ease. This part can then take a variety of input as required. This can be anything from manual signals to input from pre-written scripts written to simulate real-life

working conditions for the hardware. Lastly, this part is often also responsible for displaying the output for the simulated input. Often, the output from the hardware part is sent back to the software part for display. In this case, additional code is needed to take the output from the hardware and display it visually in some form. Once the information is input through the software, it is sent via an interface to the software.

The interface design can vary widely. It primarily depends on what requirements the hardware part has and the communication methodology being used in the system. For example, if the two systems communicate using serial port communication, the interface will implement all the code needed for this part. The interface is also designed to handle all input and output communication between the other two parts of the system. Two sets of interfaces are needed for this system. The first set is on the software side. This part is responsible for taking the simulated inputs and sending it to the hardware. This part also handles any output fed back from the hardware to the software. This happens when the software is used to display the output as mentioned previously. The other set of interfaces is on the hardware side. Like its software counterpart, this set is responsible for taking the input from the software and translating and rerouting it as necessary to the code in the hardware. It is also responsible for sending any output from the hardware back to the software.

Lastly, a hardware-in-loop system has the actual hardware part. This part is responsible for taking the input from the software side and processing it as the code for the actual hardware would. The hardware itself is often a chip or a small processor that is specifically designed for the system. Thus, this is where the user code resides. This code is usually

manually downloaded to the hardware via some other system. Once the code is downloaded, input is fed into it using the software and interfaces discussed previously. Once the input is received, the hardware part executes the user code and handles the output. The output can be displayed directly by the hardware via the use of output devices such as LEDs or speakers. Alternatively, the output can then be fed back through the interfaces to the software for display. This latter option allows for cheaper and a wider variety of display options but increases the total amount of time between input and output.

6.b Considerations in System Design

When designing such a system, a few factors must be considered. These factors all play an important role in the overall usefulness of the system and therefore must be balanced in order to achieve the best overall results possible. We discuss below some of the considerations that must be taken into account [16]:

- How closely does the system simulate an actual hardware implementation?

  The primary purpose of this system is to produce reliable data. If the data produced is either wrong or inconsistent then the system is useless regardless of how quickly and cheaply the data is generated. Thus, both the reliability of the data produced and the overall time required for a cycle must be considered. Various factors affect the time required for the system. This includes the type of operating system being used, the amount of I/O communication needed, the communication bandwidth available, the speed of the

hardware etc. Depending on the system, these factors need to be balanced. For example, some more compLex system might require a real time operating system. This is especially true in systems were timing is critical. For example, a system simulating satellites would have to provide output at exactly the right time or risk losing the satellite completely. High speed communication channels might also be required for system with a lot of communication between the hardware and software. These requirements also affect the amount of time needed for each part. Systems with a lot of communication between the hardware and software might require significant amounts of time to be spent on the interface design. On the other hand systems with low communication between the hardware and software might allow more time to be spent on the software side.

- How much hardware should the system use?

This consideration ties into the previous question. At the very least, the system must include the embedded system processor. The processor is required for the test code to be executed. Beyond that, the requirements of the system must be considered. The overall compLexity of the hardware being simulated must be taken into account. If the hardware is compLex yet relatively cheap to manufacture then it could possibly be included in the system. On the other hand, if it can be easily simulated then it possibly better to just write code to simulate it. More actual hardware in the system usually results in a better overall simulation in regards to the accuracy of the data and timing requirements. Yet, more hardware also results in a much more compLex system that is harder to maintain and costly.

- Can and should a HIL system reliably simulate this hardware?

A HIL system offers many advantages over a traditional hardware testing system. For example, a HIL system offers low cost tests. Since the actual hardware is not used, tests are very cheap and even if the system breaks, it is easily and cheaply replaceable. This also indicates the viability of repeated testing. The same input can be fed in multiple times to test the system repeatedly. Most system require multiple rounds of testing as existing code is modified and new code is added in. HIL systems can also be reliable. With close monitoring of the code written and precise control of the hardware, a high degree of reliability can be guaranteed. The main reason to not use such a system is that ultimately this system simply simulates the hardware and therefore exact timing and results can never be guaranteed. So, HIL testing is often used as a first round testing method. Once the code is working reliably in a HIL system, it is transferred to the real hardware for final testing. This option takes advantage of the cheap and repeatable testing offered by an HIL system during the early stages of development when bugs are most prevalent while taking advantage of the exact timing and result information offered by the use of actual hardware in the latter part of development.

In Section 7 we describe our own system. We describe the implementation of the various parts and how they relate to the design described previously.

7. SYSTEM DESIGN AND IMPLEMENTATION

In this part we will describe how our system was designed. We will describe the software, interface and hardware parts of it and talk about why some of the design choices were made. Since this author's work dealt primarily with the software side, we will emphasize the software side and its organization heavily and briefly go over the interfaces and hardware.

The goal of the system, as described previously, is to show an example of a cheap and reliable HIL system that can be used to simulate the workings of a much more complicated piece of hardware. In our case, we simulated a traffic light. The traffic light was chosen since it is a piece of hardware that is quite common and has the ability to react to input (traffic) and show output (the color of the light). The software and interface parts are written mostly in C#. The simulation hardware used is a TMS320LF2407 EVM.

7.a System Setup

The system is currently setup as shown in figure 3 below. At this time, the example system that we have created simulates two traffic signals responding to simulated traffic passing by on roads. Depending on the number of cars waiting and the amount of time passed, the system switches the traffic signals' colors.

The system is currently designed to run on the Windows XP operating system. Due to the fact that a traffic signal does not require very precise timing, we chose to not use a real time operating system. As we discuss later, systems requiring much more precise timing should

use a real time operating system in order to guarantee a response within a certain amount of time. This choice also helps in using this system as a teaching tool. Most students are familiar with the windows line of operating systems and thus are more comfortable programming in it. Also, there are a lot of tools available for developing software on the windows operating system. Lastly, time and financial constraints made windows XP the best choice.

The system is very small and therefore the minimum memory requirements for Windows XP suffice. The code and simulation graphic that is displayed on the screen is written in C#. The reasoning behind choosing to use C# is explained later in Section 7.d where we cover the software part of the system. In our case, we used Microsoft Visual Studio 2005 to develop this software. The students can use this same software for creating any code that they want to run on the host machine. In order to write code for the EVM any system capable of compiling C code is usable. We used Code Composer for writing software to run on the EVM described next.

Attached to the host machine is the TMS320LF2407 EVM. In this part we had a choice of using either an embedded processor or a processor simulator. We chose to go with the former since it provides a more realistic test bed for the software. Also, processor simulators add a greater lag time between input and output. We wished to avoid this lag time and therefore went with the embedded processor. This EVM is what the test code is downloaded to. The EVM here simulates the traffic light signal. Theoretically, once testing via this system is complete, the tester can unplug the EVM and plug in an actual traffic light signal. They can then download the code and have it correctly operate the traffic light signal. The host machine and

the EVM communicate with each other via the serial port on the host machine and on the DSP. The system does have interfaces built on both the host computer side and the DSP side to translate the signals as necessary as the two systems communicate. In between the EVM and the host machine we use a JTAG emulator.

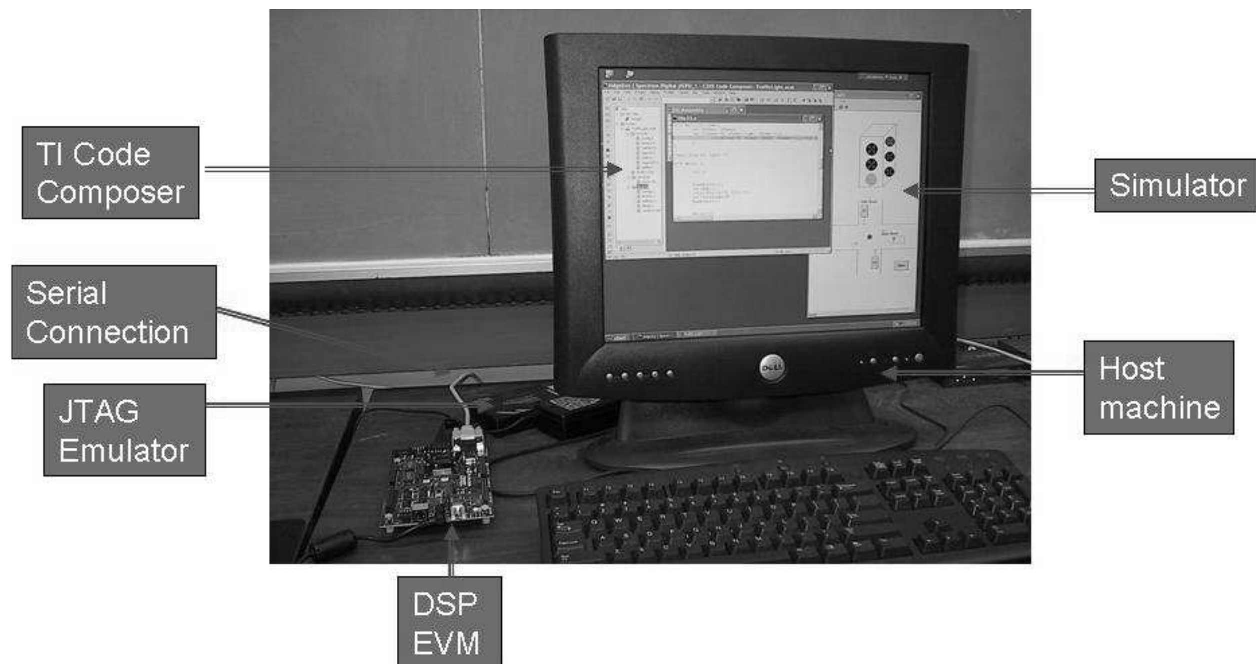

Figure 3: Picture of the System [16]

7.b Hardware

Before we go into details about how the software for the host machine works, we first discuss the attached hardware and communication methods used between them. The DSP EVM module we use to simulate the control signals is the TMS320LF2407 EVM. Figure 4 shows

a block diagram of the EVM. This EVM has various noticeable features [6]. The EVM normally operates at 30 MIPS with 128K words of zero wait state memory. It also has 16 channels of 10 bit on chip Analog to Digital Conversion with auto sequencer. It comes with User Switches and LEDs. As we discuss later, these LEDs can be used to output data showing the traffic lights' colors at every given second. However, they are not perfectly designed for our purpose and therefore we use the host computer as the output device for the light colors. Lastly, the EVM has a 5 volt power input, (onboard 3.3 volt regulators). The EVM has some major interfaces including the target ram, analog interface, the LEDs and the RS232 interface. Overall these features make the TMS320LF2407 EVM a perfect option to use as our hardware simulator.

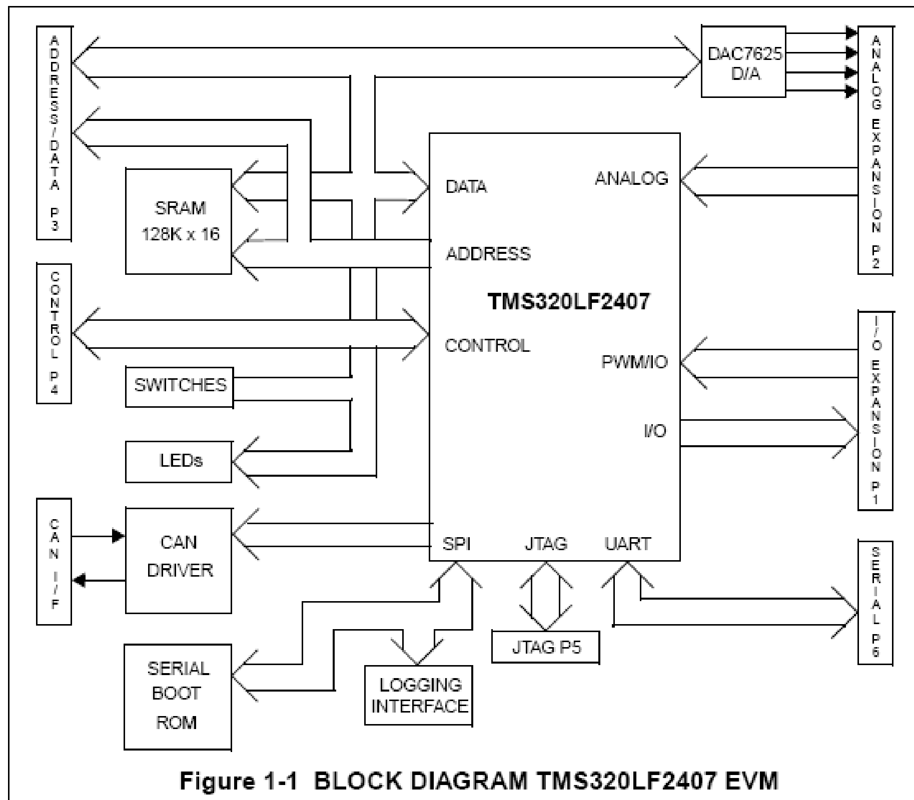

Figure 4: Block diagram for the TMS320LF2407 [6]

For the communication between the host computer and the DSP we considered multiple protocols. However, ultimately we decided to use RS-232. RS-232 is one of the most common asynchronous serial line standards. We chose this standard for several reasons. Chief among them is the fact that RS-232 is simple. We want our system to be as simple as possible in order to facilitate maximum usage and understandability. RS-232 only requires 3 wires. One wire is for ground and two are for communication going either way. RS-232 is also universal. Along with simplicity, universality is also important to maximizing usage. To that end, serial ports are available on almost every embedded processor and PC. Therefore, parts can quickly be switched out without worrying about rewriting the interface. Since serial communication is an old protocol, this interface can be updated in the future as more modern technologies such as USB become more common among embedded processors. We do understand that there are certain limitations to using RS-232. Chief among these is the slow speed of RS-232 compared with technologies such as USB. However, we believe that the speed is fast enough for our application and any significantly higher speed would be wasted since we are not using a real-time operating system.

7.c Interface

For this system we built all the interfaces ourselves. Since the DSP does not have an operating system of its own, we developed our own communication methods. Thus, we developed our own device drivers for the system.

Our goals for the device drivers were to hide as much of the details as possible from the tester while providing an API whose terminology was consistently used through the design. We believe that the majority of the time for the tester should be used actually writing software to test the hardware itself instead of having to worry about the communication channels between them. This will maximize the productivity of the tester while also maintaining a uniform and well-developed communication channel that can be reused. By taking out the development of the device drivers, we can let the tester simulate whatever hardware they want on the DSP simply by changing the code they put on the DSP itself and by modifying the C# code on the host computer end. Thus, this system can be used for multiple purposes beyond the traffic light signal implementation we describe later.

The device drivers are currently written in multiple languages. The ones on the host PC side are in C#. These are the ones responsible for sending the input received from the GUI to either the user code or to the DSP. The ones on the DSP side are written mostly in C with some parts being in assembly. The update to C# on the PC was made to make it conform with general update we made to the rest of the PC side code. Per general device driver design guidelines, we have designed the device drivers with two layers. General device driver design calls for there being two layers to device drivers: one that does the actual I/O and the other that acts as an API for the developer [20]. We have followed this principal in our design. The lower level device drivers work with the port and bits directly. The upper layer provides more abstract function calls that subsequently use the lower level calls to send the data. For example, in our system we have the generic SetValue(Registers register, int value) function that is used by the

system to set a value of a register. The SetValue(Registers register, int value) function then calls the SendData(register, value) function that actually writes that data to the serial port.

7.d Software

The software side is where the primary contribution of this thesis lies. It is currently written entirely in C #. This version of the software is an update of a previously written version that was written entirely in the C programming language. We chose to update it to C # since object oriented programming has become much more prevalent nowadays and it provided other opportunities for improvement. By switching to an object oriented language we can get efficiency comparable to languages such as Fortan while saving time due to ease of code reuse and maintenance [9]. Also, the previous version, while working correctly, was poorly organized. Many of the sections of code controlling the hardware, software and interfaces were mixed together. So, this version also has been heavily reorganized to make it fit better within the object oriented scheme. Using an object oriented language was the logical choice for an update since it allowed us to easily separate and merge various sections together. Lastly, C # was specifically chosen because it provided the use of delegates. A delegate is generally defined as one entity in a system that allows another entity in a system to carry out work on behalf of the former [21]. This feature of C# is used in creating a framework for the user code.

The software is currently entirely event driven. We believe that using an event driven system is the best method since this code represents hardware that is continuously reacting to outside input. Using events allows the user to interactively change the amount of cars on the

road. It also allows them to see how the output changes dependent upon the number of cars on the road. Without the use of events, the system would have to be restarted every time the user wanted to change the amount of traffic. The entire system also runs on a clock. The clock by default is set to tick once every second. However, this can be adjusted for testing purposes. At each tick of the clock, the user code is executed to determine the current color of each traffic light. The other option we considered was to have a script input. However, this option was not satisfactory because it would have required the tester to arbitrarily choose the amount of input traffic even before the simulation started. It would also prevent them from reacting to the output while the simulation was running.

The code is organized using the Model-View-Controller (MVC) design pattern. In the MVC design pattern, the entirety of the code is separated into three parts: the model, the view and the controller.

The view portion of the MVC design pattern is responsible for taking the data outputted by the model and rendering to a fashion that the end user can understand and, if necessary, interact with. In many applications, the view is simply a user interface. This also happens to be the case in our system. In Figure 3, we show a picture of the simulator on the host machine's screen. This is the view portion. It takes the data outputted by the user code and renders it into the changing traffic light signals. The user can thereafter interact with the view by manipulating the simulated number of cars on the road.

Figure 5 is a picture of what the user interface currently looks like. This interface appears at the start of the program and can be used by the tester to dynamically put in input while the program is executing.

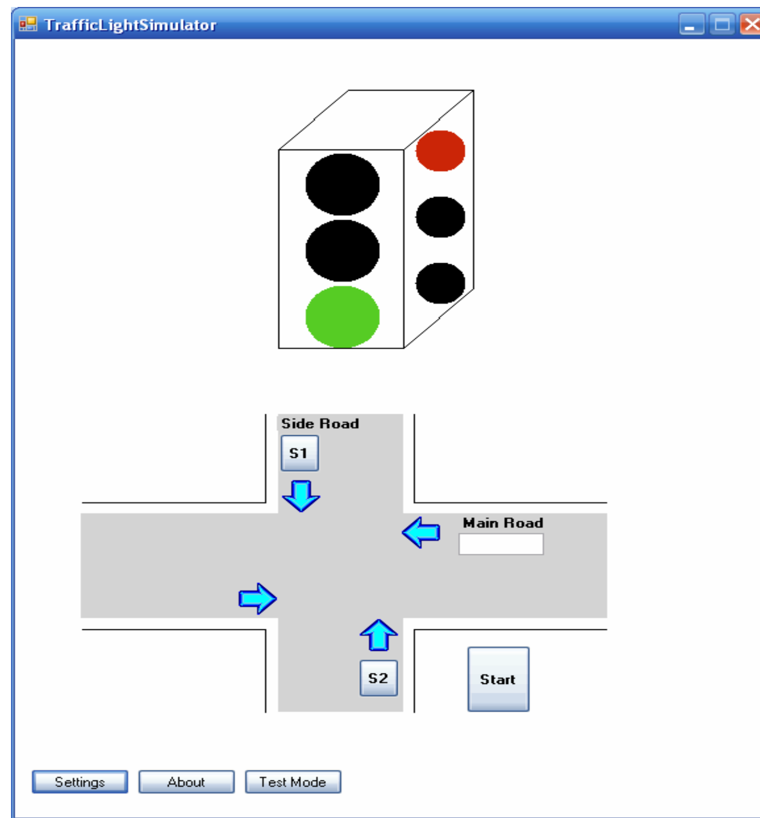

Figure 5: User Interface

Figure 5 shows all the necessary functionality currently implemented to allow the tester to test the traffic light. The image above is the initial state of the system. In this simulation, the image of the traffic light simulates how a real traffic light would behave if it were used. The color of the traffic light is dependent upon the result of the user code in the model portion of the system (discussed later).

The buttons labeled S1 and S2 control traffic on the side road (the road going north/south) in the system. Each click adds a car on that road. The main road (the road going east/west) also has a traffic monitor. For that road, a data input box is used to input the traffic density. Data from both of these traffic monitors is used to by the user code to determine the

color of the light. The program is started by clicking on the start button. Once started, the same button can be clicked to halt execution as well.

The Test Mode button in Figure 5 envelopes the host side simulation functionality of the system. If that button is not clicked then the simulation expects the user code to have been already downloaded to the DSP module and the module to be connected to the appropriate serial port on the computer. In this scenario, the system sends all user input to the serial port for transmission to the DSP. It also listens to the serial port for output back from the DSP. Upon reception of any data, the system tries to decode it. If the decoding procedure is successful, the system triggers an event and sends the data to the control layer for parsing and (if necessary) rendition onto the view layer.

Clicking on the Test Mode button triggers the host side simulation system of the program. Figure 6 shows the user interface for the test mode scenario.

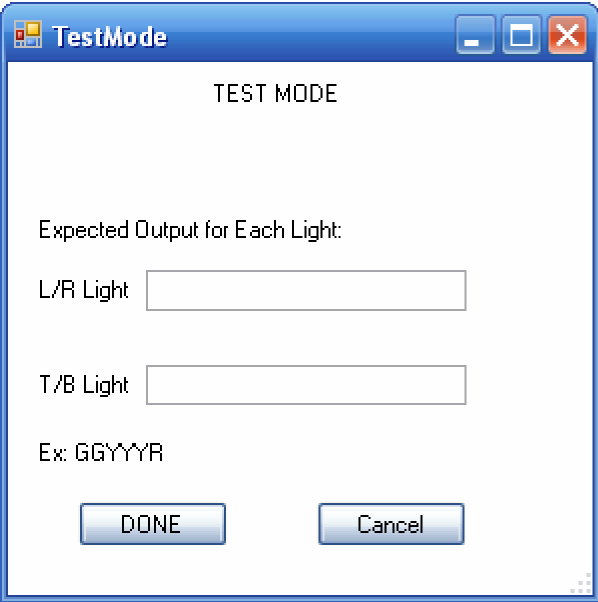

Figure 6: Test Mode User Interface

The test mode system allows the tester to specify what the output of the lights should be at each clock tick. For example, an expected output of GGYYYR would indicate that the light should be green for the first two clock ticks, followed by yellow for the next three clock ticks and red for one clock tick. This string of characters is expected for both the L/R light and the T/B light. This system is an example of a small finite automaton being used to test the system. When designing the test system we had considered using Lex to create a much more complicated and abstract automaton that could be used to create a much more thorough testing system. However, in order to keep the system simple we decided to go with a character sequence testing system instead. As we will discuss later, Lex can be used as a method of building an automaton based testing system. As we will discuss in Section 9.a, beyond just showing the output either through the host computer or the DSP itself, this system will be able to automatically test to see if the output is correct. It will do so by creating a link list of what the light color should be at each second. It will then check the color of the lights at each second against the expected output. This will allow the user to accurately determine exactly when the problem occurs thereby allowing them to more quickly discover the cause of the issue. Clicking the Done button creates the testing link list in the system. It can then be used during execution for validation purposes.

Next, we will discuss the controller portion of the MVC design pattern. The controller acts as a middle layer between the model and the view layers. It is responsible for triggering changes in the view layer depending on the data received from the model layer. In web-based systems this is the actual application code underneath the HTML output. In our case, the controller layer is represented by the simulation code. This layer has multiple parts. As we will

discuss next, the system gives the user the option of either simulating the entire system on the host computer or using the DSP EVM to download the code.   Thus, the controller layer is split into two parts: one controlling the view layer when the DSP is used and another controlling the view layer when the host computer is used solely. Both parts essentially do the same job. The chief difference is in how the input from the view layer is redirected and how the output from the model is translated before being sent to the view layer for rendering.

In our system, we chose to leave the decision on whether to use the host computer or the DSP to the user. This is controlled by the Test Mode button discussed previously. Setting up a test system triggers a Boolean in the system that determines whether or not the system is in test mode. We believe that having a single Boolean value to determine the operation is the best and simplest method. This allows for easy manipulation of the system in the future. Instead of having to deal with multiple variables and inputs in order to determine the usage of the system, one single variable determines the entire operation. If the Boolean is set to false, the system assumes a normal host machine and DSP connection and sends all output to the serial line. If the Boolean is set to true (in other words, the system is in test mode) then the system redirects all input to the user specified functions. Shown below is the code for the operation of the test mode button.

```
private void testModeClicked(object sender, EventArgs e)
        {
            string lrInput;
            string tbInput;

            if (!testMode)
            {
```

Figure 7: Host System Test Code part 1

```
            TestMode t = new TestMode();

             t.ShowDialog();

             //buttonStart.Text = "Simulate";

             lrInput = t.lrValue;
             tbInput = t.tbValue;
             testMode = true;
             buttonTest.Text = "Cancel Test";
             userCode.setExpectedLightVals(lrInput, tbInput);

             userCodeThread = new Thread(new
             ThreadStart(userCode.startSimulation));
             userCodeThread.Start();

             if (userCodeThread.IsAlive)
                 MessageBox.Show("thread is alive");
            // MessageBox.Show("testmode is true");
        }
        else
        {
             buttonTest.Text = "TEST MODE";
             testMode = false;
             userCodeThread.Abort();
        }

    }
```

Figure 8: Host System Testing Code part 2

This method creates a thread that runs the user code and sends it the expected values for the results from the test mode GUI interface described previously. If the system is already in test Mode then clicking on the button cancels the test mode. If the system is not in test mode then the button activates the test mode and sets the Boolean `testMode` to true thereby signifying that the host machine will be used by itself.

As mentioned previously, once the application is started, the entire system runs on a clock. By default, the control level checks for new inputs once every second. However, this timing can be adjusted. To use the clock feature, we have implemented a hidden clock object in the system. The clock itself is an event driven system. At each tick of the clock it calls the

`timer1_Tick(object sender, EventArgs e)` method. This method then checks for new input and either reroutes it through to the serial line or calls the user function as needed.

Lastly, the model portion of the code represents the actual data that is being accessed. In many web applications, for example, the database or XML files that are being accessed are represented by the model abstraction. In this system, the model is represented by the user code. It is what is accessed at the very core. It takes the input data and calculates what the output should be. Therefore, it acts as the "data" that would normally be accessed in a web-based application system.

Below is a sample of what the user could create in order to test the system. The test code below assumes that the test mode functionality described previously has already been activated.

```
using System;
using System.Collections.Generic;
using System.Text;
using System.IO.Ports;
using System.Windows.Forms;
using System.ComponentModel;
using System.Data;
using System.Drawing;

namespace NewTrafficLight
{
public delegate void UpdatedEventHandler(int i);

class CSampleUserCode
{
      public int LRSensor;
      private FormTrafficLightSimulator temp;
      public event UpdatedEventHandler LightUpdatedEvent;

      public CSampleUserCode(FormTrafficLightSimulator t){
              temp = t;
      }

      int i = 1;
      public void startSimulation()
      {
```

Figure 9: Example User Code Part 1

```
            //put initialization code here

        }

        public void updateLight(int val)
        {

            LRSensor = val;

            if (LightUpdatedEvent != null)
            {
            LightUpdatedEvent(i);

            if (i < 4)
                i++;
            else
                i = 1;

            }
        }


    }
    }
```

Figure 10: Example User Code Part 2

The above code is setup to show the basic structure of what the user code should look like. The above code does not use the traffic density to compute what light should be on. Instead, it simply updates the lights every second as a demonstration of how the system would work once all the input data is taken into account. It is assumed that once this code is used by students, they would finish implementing the whole thing.

As was described previously, the user code is run as a thread within within the main code. Once the test mode option is chosen, the system automatically executes the user code. We chose to setup the user code as a thread to keep with the event driven design philosophy. Once the start event is triggered (namely by clicking on the Start button), the system executes the appropriate thread.

The user code currently needs only two functions. The `startSimulation()` function and the `updateLight(int val)` function. Together, they are enough to make the user code appropriate interact with the light and update the light colors as necessary. Samples of both of these functions are shown above. The startSimulation() function does not necessarily need any code. It is simply there as a start point for the thread. This gives the user the fLexibility to add to whatever initialization code they deem necessary. To output the information back to the control, the user needs to use `updateLight(int val)`. That function takes whatever output is computed and sends it back to the control layer. The user is also free to add in whatever other functions they deem necessary.

8. EVALUATION

We believe the work presented above provides a step forward in the field of embedded system testing. The work that we have done not only provides a conceptual framework for other systems to be developed, but also shows a working system that uses the concepts and provides concrete results. Therefore, its merits should not be based solely on what the concrete example does. Instead, its impact should be looked at based on the impact it can have on other researchers' work.

This work stands out due to the fact that it uses both hardware and software. Much of the current work that is going on right now relies too heavily on simply simulating the hardware without much actual hardware being used [2,15]. This creates various problems that we described previously. By using a HIL system we solve many of these problems while still retaining the advantages of using a simulation.

This work also shows that a HIL system can be used effectively in an educational setting. By making various design choices that emphasize cost reduction and a low learning curve, we present a system that is not only effective but also very user-friendly. Various educational facilities can use the concepts presented to create cheap systems that students can pick up very quickly and use with ease.

Therefore, while there are other other systems that either emphasize a certain field or a certain methodology, our system can be expanded upon to various fields and methodologies. In section 10.a we describe various extenstions that can be made to the system to make it even more realistic and abstract while retaining the strong points it currently has.

9. CONCLUSIONS

In this thesis we have covered a wide array of issues relating to hardware testing systems. We have not only only discussed current hardware testing methods but also implement many parts of a software based hardware testing tool.

We have first discussed the current hardware testing methodolgy. We have given full descriptions of how they work and where their weak points are. We have then proposed our system which helps solve a lot of these problems. In that secion we have given an overview of how everything will work and talked about why our solution works.

The system we have presented offers a low-cost, secure, and reliable alternative to full hardware testing. Our system has concentrated on being user-friendly and cheap to produce. We have made it specifically to have a low learning curve while providing full functionality to test hardware. We have talked about the thought processes and goals behind the system and then followed that up by giving specifics of how the system is designed. We have split this section up into three sub-sections (hardware, software and interfaces) and talked about all three in great detail.

This thesis has contributed a testing framework to the system as well as significant revision of the existing code. We have helped update the code from the prevoius language and helped debug and organize it in a much more sensible way. We have also provided functionality allowing testers to test their software completely on the host system instead of using the embedded processor. Lastly, we have created a system allowing them to input expected output and allow the system to check against it.

This work has been designed to be expanded upon in the future. It has by no means completed the implementation of the system. In Section 9.a we propose areas where future work can be done on the system. We propose some changes to the hardware that will make it a more effective system and discuss the weaknesses of such a system. We also provide possible changes to the testing framework that will make it more abstract and therefore usable in a much wider range of hardware testing systems.

9.a Future Work

While we have accomplished a lot in getting our software and hardware to work together as a combined whole, there is still much work left to do. While the work of this thesis has added some parts to the software to make it better, there is still room to not only add new functionality to the existing hardware and software, but also expand upon the existing work.

Currently, our system is geared toward being cheap and user-friendly while still operating in a manner that provides reliable data within certain time constraints. For example, we want to make sure that the lights of the each traffic light are updated for one clock cycle before the next clock cycle executes. However, we have not concerned ourselves as much with providing more real-time operation for the system. Thus, at this time we can do work on making the system approximate real-time much more closely. To do this, we have a few options we can take a look at:

- Real Time OS: As we mentioned in Section 7.a, our system does not use a real-time OS. Instead of a RTOS, we use windows XP. Use of a real-time OS would significantly

enhance the response time of our system. Not only would it guarantee that requests are carried out within a worst-case time, but it would take away much of the overhead that windows XP suffers from due to a large number of background threads.

- Faster communication method: At this time our system uses the serial port for communication between the hardware and the software. While serial ports are abundant and well known, they have downsides. The main downside is the slow speed of communication relative to more modern methods such as USB. We can work on changing the communication methods to support more modern technologies. This switch will become easier in the future as more embedded systems start incorporating these modern technologies.
- New DSP: At this time we are still using the DSP that was originally chosen for this system. While that choice was logical at that time, this DSP is no longer in production and has to be special ordered. This has caused us problems during the current year with hardware breaking and needing to be replaced. Future work needs to be done to determine the feasibility and usefulness of replacing the existing DSP with a new one. Not only would this provide us with a DSP that can be fixed or replaced quickly but could possible also provide for the new communication methods discussed previously. However, using a new DSP would require complete rewriting of all existing interface code as well as code written for the DSP. Therefore, replacing the DSP should only be done if absolutely necessary.

There is also work needed to develop the work done in this thesis. As we mentioned previously, the main contribution of this thesis was in the reorganization and update of the pre-

existing code and in adding a test frame work to the host-side simulation system.   While this work has been completed to a large degree, we can still expand upon to make it more abstract:

- Lex: Currently, the test framework is tailored toward the proof-of-concept traffic light system we have created. However, this same work can be made much more abstract and therefore better suited to handle a wider range of operations. Previous work has been done in creating more abstract framework based on finite automatons by Tripakis [19]. The best method to do this would be in using a system such as Lex.

  Lex is a software that allows users to identify tokens and associate code with that token. Essentially, it is a Lexical analyzer (hence the name Lex) generator [4]. We believe this technology can be used to create a framework using which a tester could identify the expected output and then associate code that can be run based upon the actual output. This would be an extension of the testing system we created for the traffic light system. Essentially, instead of having them give a string of expected lights, this new system would allow them to specify the expected output to whatever they want. Then, instead of having to manually account for each token in their own code, they would use Lex as the generator for this code. This in turn will allow users to choose different architectures, mappings and schedules before the actual implementation [7].

  We can foresee two issues that will come up with this concept. First, at this time Lex is not widely used. Therefore, significant care will be needed to make sure that the system is not overly complicated and well-documented. The system will need to ensure that enough capability is provided to allow the tester to write code usable in a broad range of situations while simple enough so that it can easily be picked up. The

other issue is that currently the different versions of Lex output code either in C or C++. Therefore, our code will need to be modified to handle code in these languages. This might require significant modifications of existing code as well as writing new code to handle the interfacing between the existing code and the new code.

Lastly, in the future we can expand the current system to allow for multi-user usage. At this time, the system is designed to have one DSP attached to one host computer being used by one tester. This setup was shown in figure 3. However, this can be easily expanded to work in a network setup. Using a new communication system such as USB will allow the host system to talk with multiple DSPs at the same time. Further, using the already built "user account" system in windows XP, we can setup multiple user accounts. This can also be done through most other operating systems such as various flavors of the Linux operating system. Thus, we can turn the host computer into a server. Multiple testers can then log onto this server through the World Wide Web. This setup provides numerous benefits:

- Increased tester productivity: Through the use of a networking system, multiple testers will be able to use the system at the same time. Thus, the overall product development cycle will be sped up. Instead of having to use the hardware one by one, multiple testers will be able to work at the same time. This will not only allow them to find bugs quicker but also get them to the developers quicker for fixing. This will significantly cut the product development time and therefore save time and money.
- Security: Currently, the tester needs physical access to all the hardware in order to test their code. Using the networking system, the hardware can be located in a remote and

secure facility. This will provide significantly enhanced security. Individual testers will not need security clearance and the facility can be monitored much more closely. This feature will prove to be especially useful when very expensive hardware (such as a satellite) is tested.

10. BIBLIOGRAPHY

[1] Apple Inc. http://www.Apple.com

[2] UPPAAL TRON. http://www.cs.aau.dk/~marius/tron/

[3] Embedded Systems Design. http://www.Embedded.com

[4] A Compact Guide to Lex & Yacc. http://epaperpress.com/Lexandyacc/index.html

[5] The Mathworks. http://www.Mathworks.com

[6] TMS320LF2407 Technical Reference Module. http://roinos.com/board/evm/evmlf2407a_r.pdf

[7] P. Arato, S. Juhasz, Z. A. Mann, et al. Hardware-software partitioning in embedded system design. In *2003 IEEE International Symposium on Intelligent Signal Processing*, pages 197 - 202

[8] Ali Keyhani, Gerald Baumgartner, et al. An integrated Virtual Learning System for the development of Motor Drive Systems. *IEEE TRANSACTIONS ON POWER SYSTEMS, VOL. 17, NO. 1*, Feb 2002

[9] R. G. Bruce. Distance delivery and laboratory courses in *Proc. ASEE/IEEE Frontiers in Education Conf*., 1997, Session F1B

[10] *Heath, W.S. Real-Time Software Techniques. Van Nostrand Reinhold, New York. (1991).*

[11] Larry H. Johnson and Charles M. Crocker. Cost effective Weapons system development through integrated modeling and hardware testing. In *Winter Simulation Conference Proceedings of the 25th conference on Winter simulation*, pages 1365 - 1367

[12] M. Grindal, J. Offutt, et al. On the testing maturity of software producing organizations. In *Testing: Academia & Industry Conference – Practice and Research Techniques (TAIC / PART 2006)*, Windsor, UK, August 2006

[13] A. Keyhani and A. B. Proca. A virtual testbed for instruction and design of permanent magnet machines. In *IEEE Transactions on Power Systems*, Vol. 14, No. 3, pages 795-801

[14] N. Kranitis, G. Xenoulis, et al. Zorian Low-Cost Software-Based Self-Testing of RISC Processor Cores Design, Automation, and Test. In *Europe Proceedings of the conference on Design, Automation and Test in Europe* , Vol. 1, Page 10714

[15] Kim G. Larsen, Marius Mikucionis et al. Testing Real-Time embedded Software using UPPAAL-Tron. In *International Conference On Embedded Software. Proceedings of the 5th ACM international conference on Embedded software*, pages 299 – 306

[16] Jing Liu. A Virtual Testbed for Embedded Systems Development and Instruction. Master's thesis. The Ohio State University. 2004

[17] S. B. Mobley and J. P. Gareri. Hardware-in-the-loop simulation (hwil) facility for development, test, and evaluation of multispectral missile systems: update. In *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing V*, Vol. 4027, No. 11, page21

[18] Patterson, David A., John L. Hennessy. Computer Organization and Design 3rd Edition. Morgan Kaufmann Publishers, New York: 2005

[19] Tripakis. Fault Diagnosis for Timed Automata. In *Formal Techniques in Real-Time and Fault Tolerant Systems. (FTRTFT'02)*, volume LNCS 2469

[20] Shaojie Wang *and* Sharad Malik. Synthesizing Operating System Based Device Drivers in Embedded Systems. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis CODES+ISSS '03*

[21] Longhua Zhang, Gail-Joon Ahn, et al. A Rule-Based Framework for Role-Based Delegation and Revocation. In *ACM Transactions on Information and System Security (TISSEC)*, Volume 6 Issue 3