2014

A Virtual Testbed for Embedded Systems

Shrikant Labade
Louisiana State University and Agricultural and Mechanical College, shrilabade1@gmail.com

A VIRTUAL TESTBED FOR EMBEDDED SYSTEMS

A Thesis

Submitted to the Graduate Faculty of the
Louisiana State University and
Agriculture and Mechanical College
in partial fulfillment of the
requirements for the degree of
Master of Science

in

The Department of Electrical and Computer Engineering

by
Shrikant D Labade
M.Sc., University of Pune, India 2007
May 2014

To my Family and Friends back in India

ACKNOWLEDGMENTS

I take pleasure in submitting herewith the report on “A Virtual Testbed for Embedded Systems” in partial fulfillment of the requirements for the degree of Master of Science in Electrical and Computer Engineering.

I am grateful to my co-advisor Dr. Gerald Baumgartner for his wholehearted support and valuable suggestions without which I would not have been able to complete my thesis. His deep knowledge and keen interest in the field encouraged and motivated me to come up with the present work. Thanks for explaining the concepts again and again and being patient. Our weekly meeting has really helped to get that fuel to work regularly on my thesis and imparted me with a knowledge that will help me in other walks of life as well.

I would like to offer my most sincere thanks to my co-advisor Dr. J. Ramanujam for his constant suggestions and the knowledge he bestowed upon me in and out of the class.

I am very thankful to the College of Humanities and Social Sciences at LSU for providing me the financial help during my Masters. I would like to take this opportunity to thank my manager Mr. Mark Hovey for his constant support. Working with Mr. Mark was a great experience which taught me a great deal. The flexibility you gave me at work was truly beneficial in that it allowed me to concentrate on academic activity.

I would also like to thank my committee member Dr. Suresh Rai for graciously agreeing to be on my thesis committee.

I am thankful to Newton Odgen for his help in the initial development of device drivers. I am also thankful to Ajay Panyala for his help with measurements.

I sincerely thank Kevin Hartline for proof-reading my thesis and suggesting valuable changes.

I am very grateful to my family for their constant moral support and belief in me. You’re love gave me the courage to strive and succeed in all my endeavors.

I sincerely thank the Department of Electrical and Computer Engineering for allowing me to use the resources to complete my thesis.

TABLE OF CONTENTS

ABSTRACT

Hardware-In-the-Loop (HIL) Simulation is a simulation approach in which a hardware embedded processor is connected to the simulation computer that simulates the electrical/mechanical devices controlled by the embedded processor. By using a real-time simulation computer and special-purpose hardware for connecting to the embedded processor, this method of simulation can be very precise but is costly. We are proposing an alternative method, HIL simulation with a network link, in which the device under test (the embedded processor) communicates with the simulation computer over a network connection (in our case a serial line) instead of through special-purpose hardware. We present an abstraction layer that facilitates the simulation of external devices. An earlier prototype had been developed for a 16-bit TMS320LF2407A DSP from Texas Instruments. We generalized the approach to the more advanced 32-bit TMS320F28335 DSP. We have made the changes in the DSP abstraction layer to enable more features and provide more flexibility to the programmer. For example, we introduced a shadow interrupt vector to make the simulation layer more general. We developed various scenarios to measure the performance of the system. In particular, we measure round-trip time and through-put for the communication between the simulator and the DSP. Also we rewrote the serial line drivers on the DSP to incorporate different working scenarios and to invoke the timers on the DSP for measuring the execution time. Our work helps to judge the performance of the system and to identify the application domains for this approach.

CHAPTER 1. INTRODUCTION

An embedded system is a computer system that performs a task using dedicated hardware in real-time. An embedded system repeatedly communicates with its surroundings through sensors and then responds to the surrounding using actuators. For example, in a temperature control application an embedded system reads the temperature sensor and then decides whether to turn on the air conditioner or not. An embedded system has either a Microcontroller or higher power Digital Signal Processor (DSP) as a processing core based on the application the embedded system is supposed to perform. For example, an embedded system designed to control an elevator will have a DSP, as it has to process large amount of data to control the motor driving the elevator; on the other hand an embedded system designed to control lights will have a Microcontroller since it just has to read the sensor and activate the actuator to turn on the lights. Figure 1 shows the block diagram of a general embedded system.

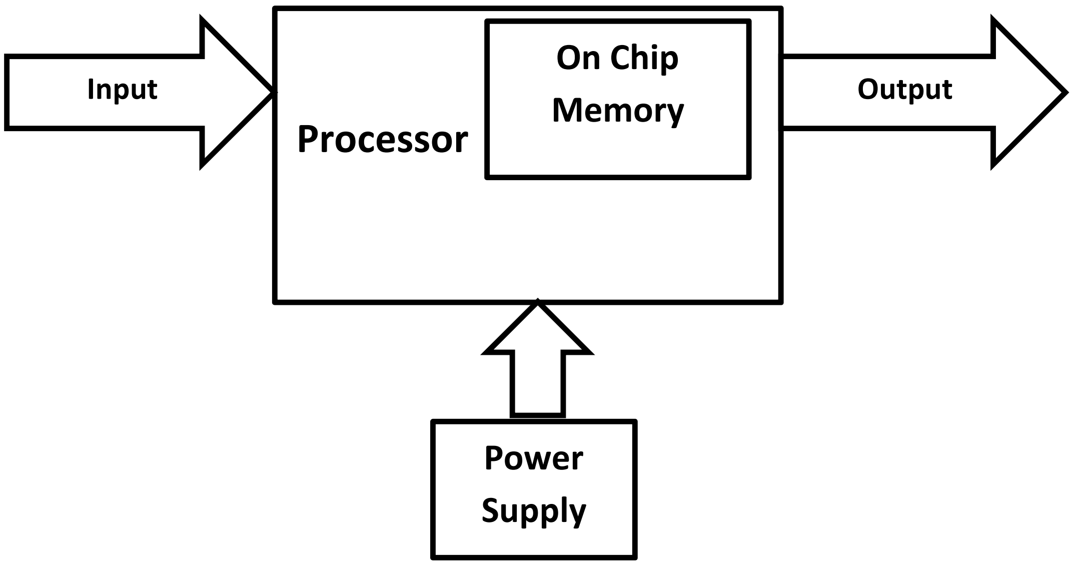


Fig 1: Basic Block Diagram Embedded System [19]

Today, embedded systems are rapidly becoming ubiquitous and more complex in nature. We can find applications for an embedded system in almost every field. Embedded systems can be found in the operation room in the form of a patient monitoring system to assist the surgeon in the surgery. They are also used to remotely control certain toys, such as cars or helicopters. Some interesting statistics state that the average consumer now interacts with embedded microprocessors up to 300 times per day and today, on an average, we have as many as 50 embedded processors in an automobile. However, the development of embedded systems is a tedious job, as they have a number of constraints. Some of these constraints include real-time operation, processing power, size, weight, power consumption, reliability and cost. Also, experts from various fields of electrical and computer engineering are needed to design electrical and computation parameters of the system as well as experts from mechanical engineering are required to design the enclosure and internal arrangement of various parts of the system. Using simulation will be a great help in embedded system design.

Simulation is used in every field to aid the process of development. For example it is used to simulate flight, automobiles, and bridges in civil engineering; it is also used to train pilots. Simulation is used when the system under development is complicated to design, hazardous and/or difficult to test, or expensive. Simulation is also used to study various parameters of the system, to check the performance of the system, to understand operation of the system and to train various personnel. Simulation can also help lower the development cost and lead time.

Several simulation approaches have been proposed for the development of embedded system software such as, Hardware-In-the-Loop Simulation (HIL), Instruction-Set-Simulation (ISS) and Software-In-the-Loop Simulation (SIL). In her thesis, Jing Liu has described a hardware in-the-loop simulation approach with a network link. The prototype had been developed with a

TMS320LF2407A DSP from Texas Instruments that simulates external electrical and mechanical signals that are controlled by the embedded processor. She had developed device drivers that allowed to access either the hardware or to communicate with the simulator. This system was developed to provide a design and development environment for embedded software that would be as realistic as possible [15].

Liu's design of the device drivers was not general enough to work for all device registers. Medavarapu [6] generalized this approach for all device registers and developed abstraction layers for both DSP and the PC side of the system. These abstraction layers helped test both the real and simulated hardware connected to the DSP. He modified the serial line driver to utilize the serial line bandwidth to allow using the system in an application that requires signal processing and voice [6].

This thesis is based on the work done by both Liu and Medavarapu. We have ported the software to a more advanced embedded processor. The system was originally developed on the TMS320LF2407A DSP, which is a 16-bit processor. We ported the system to the 32-bit TMS320F28335 DSP, which provides more functionality but also makes it more challenging to design the network communication between the DSP and the simulator. We also wrote shadow interrupts to enhance the simulation and invoked the timers on the embedded processor/DSP to measure different attributes of the system. The logic for serial line has also been changed to allow measurements for different cases. Also, we made the software more users friendly. For example, we added generic SET/GET functions to write to and read from the low-level device drivers, developed logic to select and determine the mode of operation, (for example, hardware, simulation and hardware and simulation mode), developed the logic to write either to the physical registers or shadow registers based on the mode of operation. Also, we introduced various test cases to measure the performance of the system. The work we have done is intended to help determine the performance of the system for different cases and to identify the area where this system can be used to enhance the process of development and to enhance the productivity and lead time for development.

This approach can be useful for teaching embedded systems and is more cost effective than the traditional teaching methods, as it does not require actual hardware devices but only a conventional PC and a DSP evaluation board. Also, a course based on the virtual testbed can be scaled to a larger number of students because it is not necessary to have a supervisor monitor the students in the as there is no danger of damage to the laboratory property because of the faulty software. This approach will also be useful in rapid prototyping since the software engineer can start the software development even if the hardware is not ready. This will help to develop both the software and the hardware simultaneously. For example, a company developing a washing machine can have the simulation library ready for almost all components of the washing machine. When they want to develop a new advanced washing machine, they can use the simulation library to develop the washing machine simulator in less time and start the development process thereby considerably reducing the development time.

Another useful application area is unit testing. In unit testing, the different components of a system are tested independently allowing the testing to be more rapid and less complicated. The simulator we are proposing can be used as a test harness, simulating some of the components of a larger system that communicate with the hardware under test. For example, the engineer testing the safety features of an automobile can simulate components such as the anti-braking system or steering control system of the automobile and connect the simulator to the actual hardware for safety features. This allows individual testing and development of the entire system making the development process faster.

This approach can also be used for monitoring purposes to log all the events on the boundary between the embedded processor and the outside world. For example, if the whole system is ready and running, a device driver can be developed that will send the data over the serial link to the simulator to be recorded for maintenance purposes. This stored data can be helpful to study the system behavior for improving the performance of the system or for identifying potential hazards and implement measures to avoid them. It will also be helpful for education purposes.

The thesis is structured as follows. In Chapter 2, we discuss the existing work done in this area. We mention several simulation approaches proposed and used in industry and compare the advantages and disadvantages. After that, we provide details of Jing Liu's approach and compare it with other existing methods. Also, we will talk about the work done by Medavarapu to make the system more realistic and user friendly. Then, we will talk about our work and how it is helpful to measure the performance of the system to determine its advantages and disadvantages over the others. In Chapter 3, we discuss the overall design of the system and provide the details of the protocol used in the system. In Chapter 4, we present the implementation details. In Chapter 5, we provide the measurements for different test cases to evaluate the performance of the system. Chapter 6 concludes our work and discusses future work.

CHAPTER 2. RELATED WORK

2.1 Hardware-In-the-Loop (HIL) Simulation

Over time, several methods for simulating embedded systems have been proposed. HIL simulators are one of the most used simulators in the industry. A HIL simulation has the embedded system under consideration connected to the hardware simulation computer. A HIL simulation can be very costly as it often requires dedicated hardware. Figure 2 shows the basic block diagram of a typical HIL simulator.

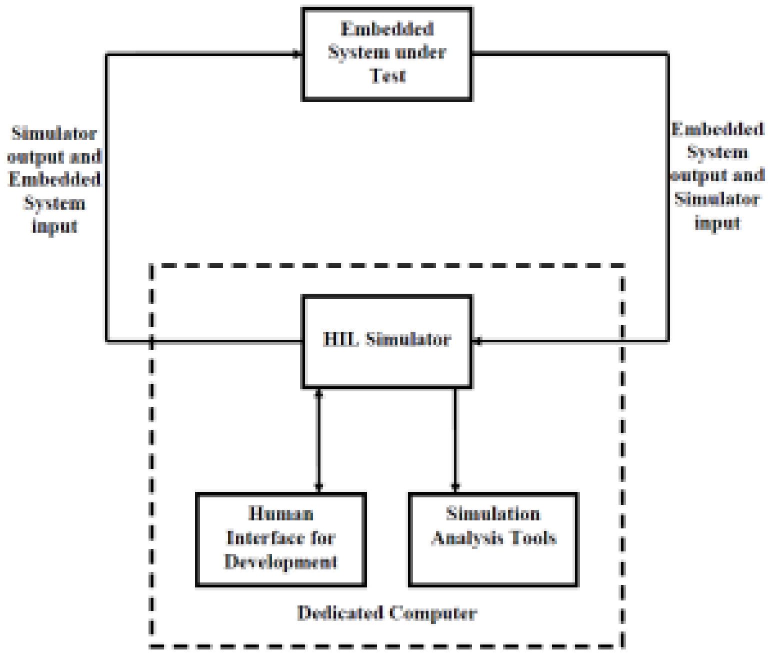

Embedded System under Test
Simulator output and Embedded System input
Embedded System output and Simulator input
HIL Simulator
Human Interface for Development
Simulation Analysis Tools
Dedicated Computer

Figure 2: Block Diagram of a HIL Simulation [6]

A HIL simulation can be very useful in measuring the timing aspects of the electrical signals as the actual hardware is being used [18], [26]. A HIL simulation can be used only when the embedded hardware is ready; hence, it can be the last stage in the development process when checking the performance of the system. A HIL simulation is very precise but tedious to implement. P. Baracos et al. are using PC based HIL simulation for automotive applications [27]. M. D. Jokie and et al. are developing “Tethered Satellite System Models” to be used in HIL simulations [20]. Refer to [2], [4], [7], [12], [13], [17], [18], [21], [26], [28] for some more applications of HIL simulation.

2.2 Software-In-the-Loop (SIL) Simulation

SIL is one of the most used and reliable methods for testing real time embedded systems. As the name suggests, the embedded software is executed on a processor different than the target processor, in close loop with the model of the plant. Normally, a PC is used to run both the embedded software and the plant model. A SIL is achieved by running an abstraction of the targeted embedded processor on the host computer. A SIL simulation is easy to implement and cost effective, as it can be executed on a regular PC and does not require any dedicated hardware. Figure 3, shows the basic block diagram of Software-In-the-Loop Simulation.

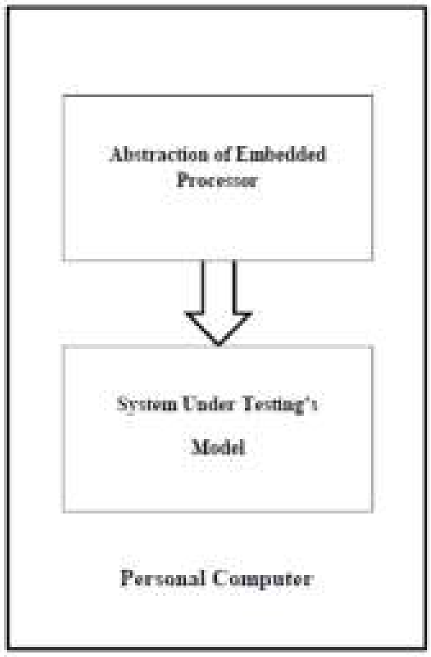

Abstraction of Embedded Processor
System Under Testing's Model
Personal Computer

Figure 3: Block Diagram Software-In-the-Loop Simulation [6]

A SIL does not represent the timing of an actual embedded processor; hence, it is less precise [15]. The abstraction can be very basic, simply to check the functionality of the software, or full-fledged to simulate the actual embedded processor.

2.3 Instruction-Set-Simulation (ISS)

In this approach, the behavior of the targeted embedded processor is simulated on the host computer [5]. As the name implies this simulation simulates the computations corresponding to each instruction of the processor. The precision of the simulation can be either cycle-level-accurate or function-level-accurate. In the function-level-accurate simulation the main focus is to evaluate the functional capability of the processor. In this case, the instruction pipeline of the processor is coarsely modeled. This type of ISS is fast but not accurate. In cycle-level-accurate simulation, the instruction pipeline and data path of the processor is closely modeled. This type of simulation is very precise but has the limitation of slow simulation speed [1]. An ISS could be implemented in several ways. Figure 4 shows the general sequence of ISS.

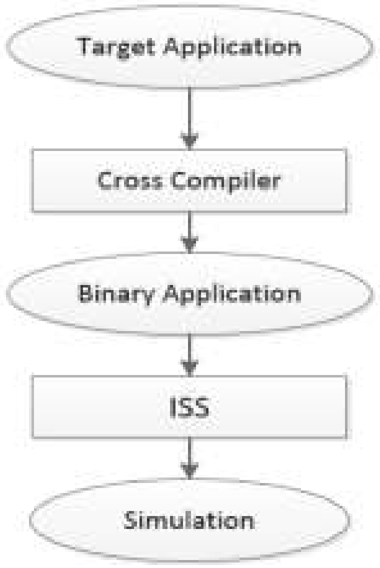


Figure 4: ISS Software Flow [1]

a. Interpretive Simulation:

These simulators interpret the targeted processor instructions. Each instruction is fetched from the memory of the host machine. The instruction is then decoded and executed. Figure 5 shows the sequence of operation.

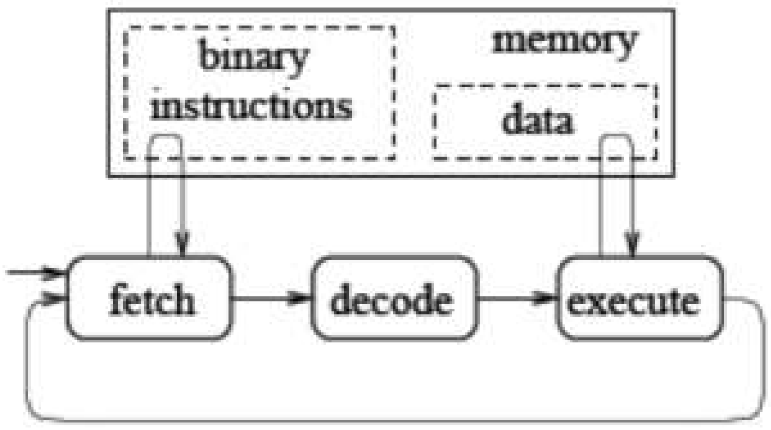


Figure 5: Interpretive Simulation [5]

This kind of simulation module is easy to implement and flexible, but has the drawback of slow speed. This slowness is caused, by the time taken to decode and execute the instructions. The speed can be enhanced by storing the decoded instructions in the decode buffer, similar to cache in computers, so that they do not have to be decoded again if they are reused. Refer to GDB [10] and TRAP [34].

b. Compiled Simulation/Static Translation:

In these simulations, the application program is translated into a new program for the host machine at the preliminary stage of compilation. This is done in order to save time spent fetching, decoding and executing operations. This simulation has a disadvantage of not simulating the processor conceptually. This simulation is also not as flexible since the entire program has to be known at compile time. This simulation also does not allow the program to be changed at run time. Hence this method is not suitable for applications which require dynamic modification of the program. Figure 6 shows the block diagram for compiled simulation. Refer to [3], [24] and [25] for techniques to improve compiled simulation.

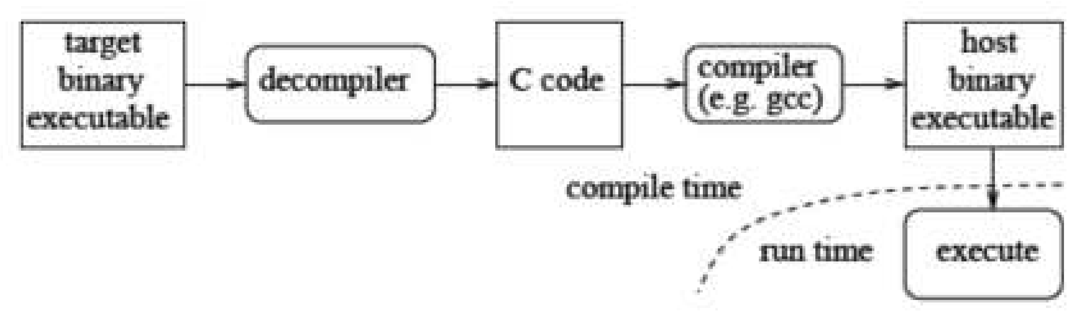


Figure 6: Compiled Simulation [5]

c. Dynamic Translation:

This simulation overcomes the disadvantage of not being allowed to change the program at run time that plagues compilation based simulation. Like interpretive simulation, the targeted instruction is fetched from the memory at run time. The instructions are then decoded on their first execution. Once the instruction is decoded, the simulator translates the instruction into a different representation and stores it in cache. If the instruction needs to be executed again, it is fetched from cache. If the program is modified at run time, the cache is discarded and the changes are incorporated [5]. In interpretive simulation, the slowness is caused by fetch, decode and execution operations for each instruction. This is overcome in dynamic simulation by using cache for repeated instructions. This type of ISS combines the advantages of both interpretive and compiled simulation. Figure 7 shows the general block diagram for dynamic simulation.

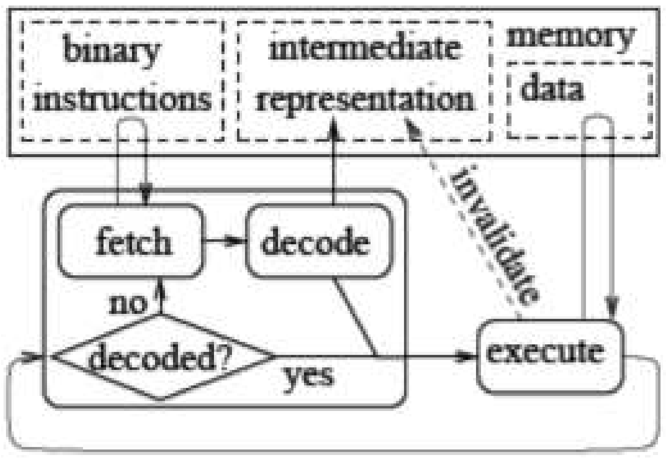


Figure 7: Dynamic Simulation [5]

2.4 Validator Simulation

The validator simulation engine is a discrete event simulator that models the operations of the system under test as discrete events in time and also takes the platform specifications into account. The platform specifications, such as operating system (OS), timers and the communication bus are the plug-ins of the validator simulation. As in SIL simulation, the validator also does not require a target platform for execution of the embedded software. It also keeps the simulation of the embedded system that is run on a dedicated hardware in HIL and the controller task that is run on actual embedded system in HIL simulation, separate. This simulation fills the gap between SIL simulation and HIL simulation by offering accuracy as good as that of a HIL simulation and cost and speed advantages of a SIL simulation [15]. Both of the simulations can either run on the same computer or on different cores or on different computers.

The Validator tool performs time-functional simulation of control software and its execution platform in closed-loop with Simulink plant models. This tool is based on a discrete event simulator, it executes the application software on a host platform simulating the passage of time according to execution times of the application code and communication times pertaining to a given embedded processor. It also simulates the functional behavior of the operating system and hardware components to the level of abstraction this enables capturing significant timing aspects without executing detailed hardware models. Figure 8 shows the architecture of the Validator simulator.

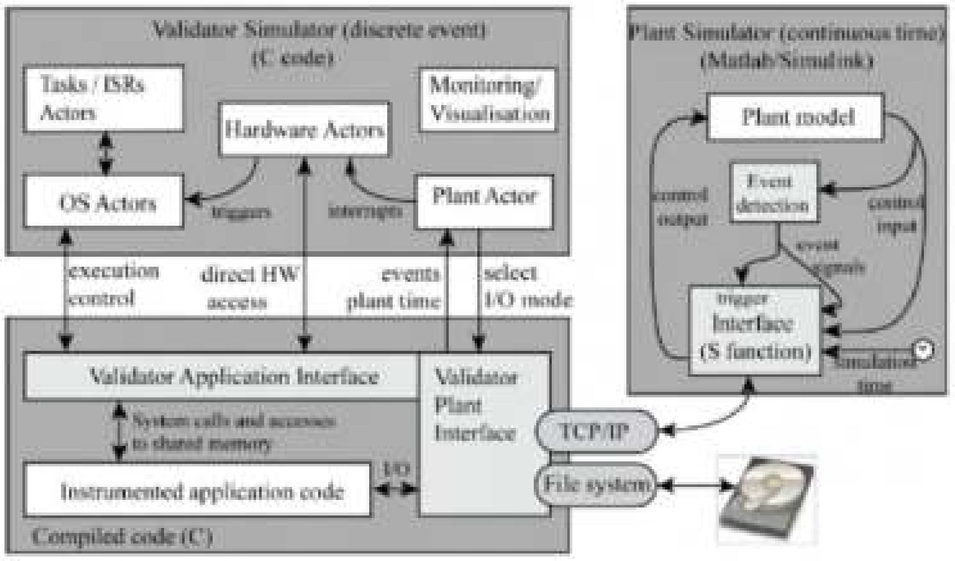


Figure 8: Architecture of the Validator Simulator [29]

2.5 Full System Simulation

Full system simulation is valuable to product developers since it is capable of running the exact binary embedded software, including the operating system, on a fully simulated hardware platform. Most simulation environments do not provide full system simulation nor do they support hardware simulation. Full system simulation helps to get rid of cycle accurate simulation, which is very slow. SimSoC is a simulation software tool developed on the concept for full system simulation [5]. It makes use of SystemC to model different hardware components in the targeted hardware and Transaction Level Modeling (TLM) abstractions to model the communications. Refer to [30] and [8] for SystemC and TLM. Figure 9 shows the architecture for SimSoc Simulation.

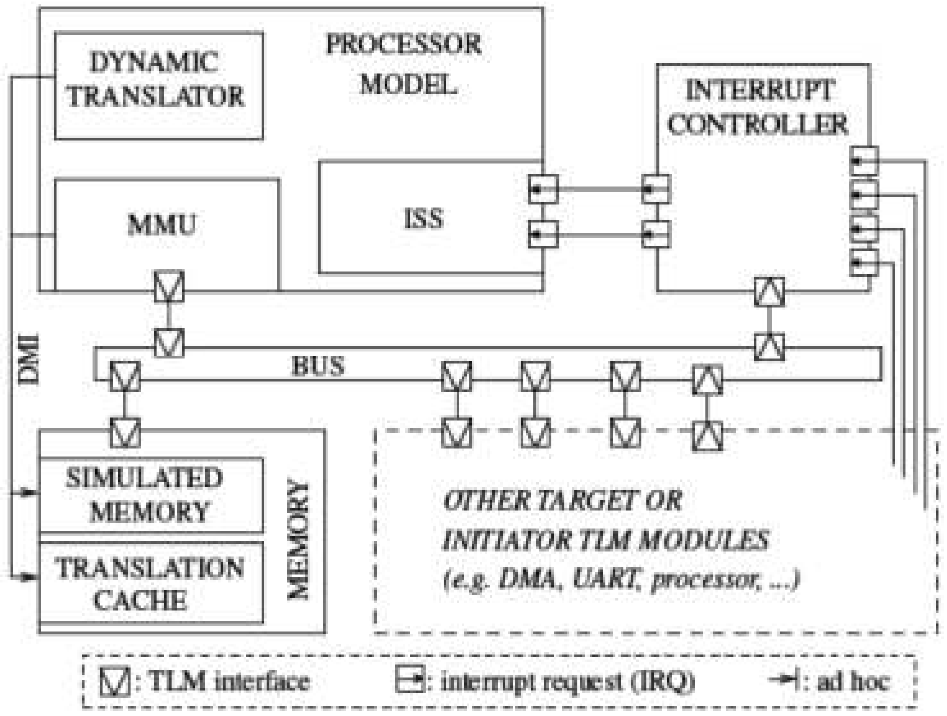


Figure 9: SimSoc Architecture [5]

2.6 Hardware-In-the-Loop with Network Link Simulation (Our Approach)

This approach fills in the gap between the HIL and SIL simulation. This approach is cheaper than traditional HIL simulation and ISS simulation, while being significantly more accurate than SIL simulation and the Validator approach. We are using a serial line communication between the embedded processor and host PC as a network link. The application code runs on the actual embedded processor and the simulation of the peripheral electrical/mechanical devices runs on the host PC.

A virtual testbed approach was proposed by A. Keyhani for the design of a permanent magnet machine. The software uses a combination of lecture notes and computer aided design (CAD) [2]. G. Baumgartner and A. Keyhani proposed a virtual testbed for the instruction and design of embedded systems [9]. The prototype was developed by Jing Liu [15] and consisted of a

PC running on Windows XP and Texas Instrument's DSP TMS320LF2407A. She had developed two HIL simulators for teaching purposes, one for a traffic light and one for an answering machine. The simulators talked to the DSP over a serial line. Medavarapu [6] further developed the protocol and the software to make it easier to use by developing more sophisticated device drivers for all devices on the DSP. For example, Medavarapu improved the communication between the PC and the DSP. Our thesis is an extension of Liu and Medavarapu's work. We ported the existing software over to a more advanced DSP from Texas Instruments to take advantage of the improved hardware. The earlier processor was 16-bit DSP, while we are now using a 32-fbit floating-point DSP. We provided more features and took measurements to demonstrate the performance of the system.

CHAPTER 3. DESIGN

3.1 System Setup

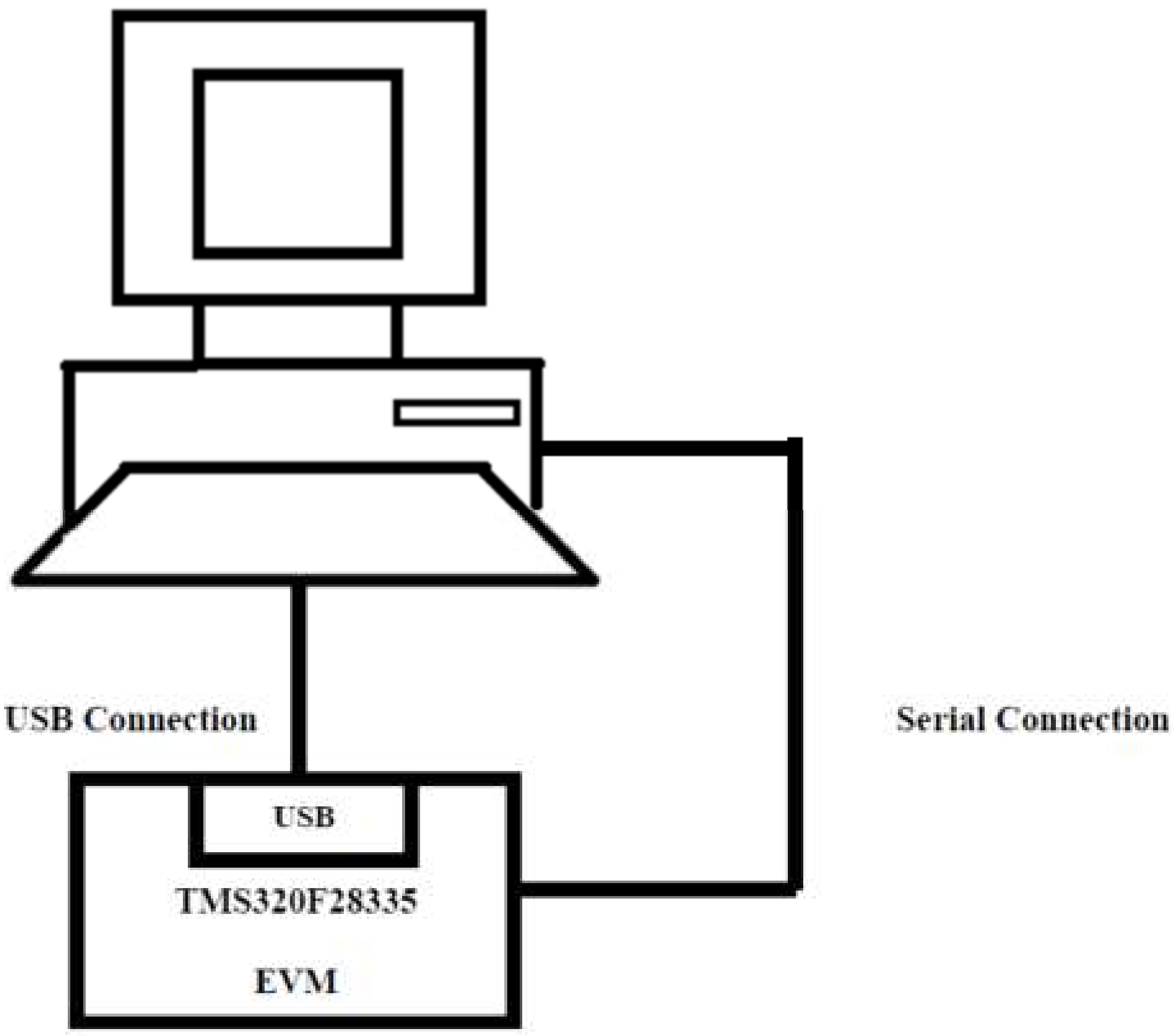


Figure 10: Block Diagram of System Setup (modified from [6])

As shown in the above figure, the system consists of a PC running on Windows 7 and evaluation module (ezDSP) for TMS320F28335 from Spectrum Digital Inc. This module allows developers to evaluate the performance of the DSP for their application. This module facilitates interfacing various devices to the DSP. This module has an onboard JTAG connector to provide the interface to emulators with assembly language and ‘C’ language. A Mini USB to USB cable is used to connect the evaluation module to the PC. The Serial port on the DSP is connected to the

PC's COM port 1 using a serial cable. Code Composer Studio (CCS) of Texas Instruments is used to program the DSP and to debug the software. The simulator, which is a GUI application, is developed using C# and runs on the PC. Figure 11 shows the actual set up of the virtual testbed system.

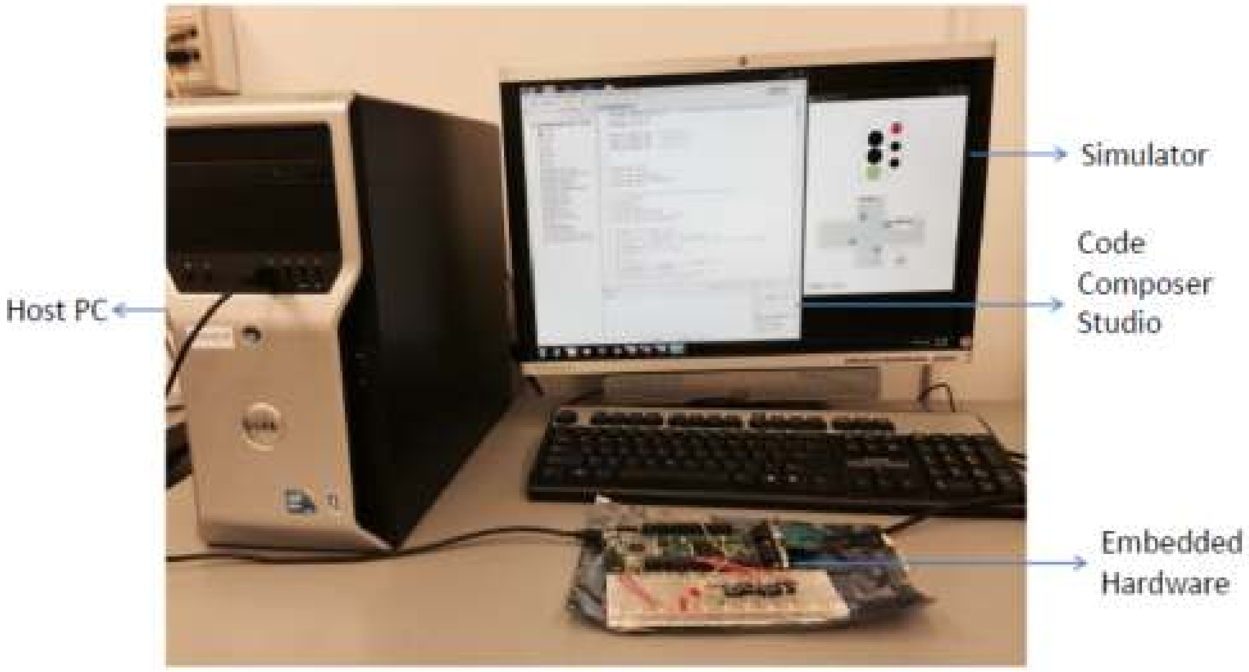


Figure 11: System Setup for Virtual Testbed

3.2 Architecture of the Embedded Processor

The embedded processor TMS320F28335 is a 32-bit floating point DSP. It offers highly integrated, high-performance solutions for demanding control applications. It can operate up to 150 MHz and has 88 general purposes input/output. This DSP has three 32-bit timers with pre-settable periods and with 16-bit clock pre-scaling. Also, it has two serial line interfaces with 64K programmable baud rates. The DSP has eight masked external interrupts; this enables it to address multiple asynchronous events. The fast interrupt response with automatic context saving of critical registers reduces the latency for asynchronous events [31]. The DSP features 68K bytes of on-chip RAM and 512K bytes of on-chip Flash Memory. The code module security (CSM) feature prevents access/visibility to on-chip memory to unauthorized persons. The DSP uses multiple buses to move data between memory and peripherals and the CPU. The multiple bus architecture allows the DSP to fetch an instruction, read a data value and write a data value in a single cycle [31]. Figure 12 shows the functional block diagram for TMS320F28335 DSP.

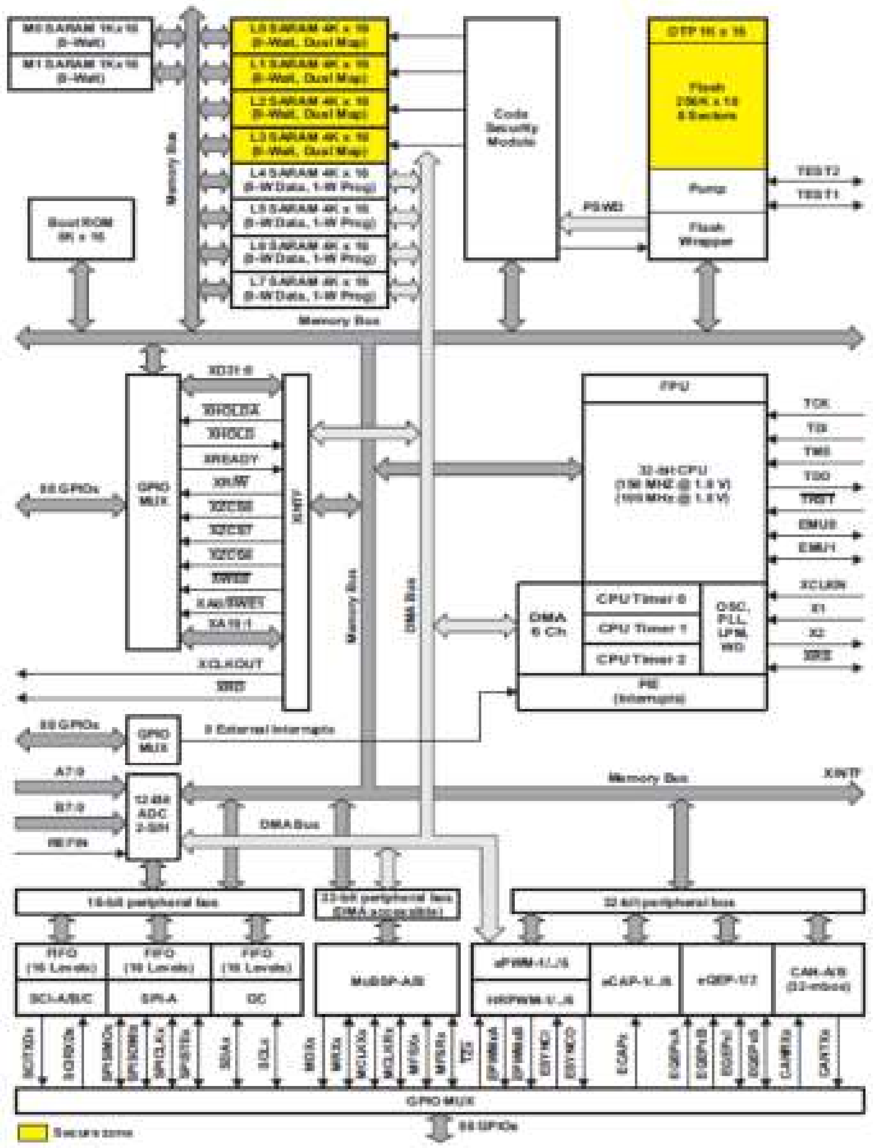


Figure 12: Functional Block Diagram of TMS320F28335 [31]

3.3 Architecture of the System

As previously stated, the system is analogous to HIL with an addition of a network link. Here the hardware is the evaluation module (EVM) for the embedded processor. The EVM is connected to the conventional PC using a serial cable. The embedded processor does not run any operating system, whereas the PC runs an operating system. Code Composer Studio and the simulator that simulates the external devices connected to the DSP runs on the PC. The status of the simulated devices is sent to the DSP over the serial line. The DSP then runs the control logic and sends the updated values for the simulated devices to the simulator running on the host PC. Users can set the mode of operation to hardware or simulation mode or both. If simulation mode is enabled, the DSP updates the values of the shadow registers; if hardware mode is enabled it updates the physical registers on the DSP. Figure 13 shows the layered architecture of virtual testbed.

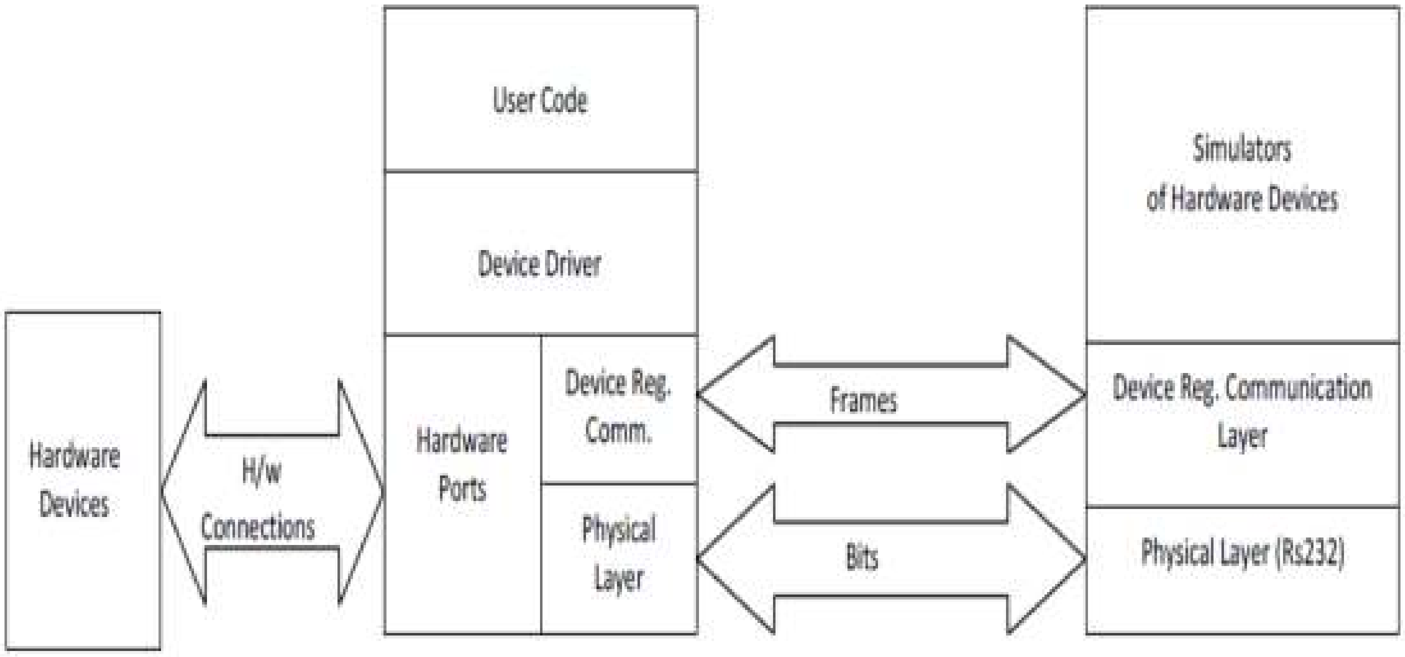


Figure 13: Layered Architecture of Virtual Testbed

As the embedded processor does not run any operating system and the host PC does run an operating system (Windows), this system is a cross platform network system. We have organized the network link between the embedded processor and the host PC into a layered hierarchy. We have two layers: the physical link layer and the device register communication layer. Ideally, to check the real time performance of an HIL simulation, we should use a high performance computer system. However, due to the limited availability of equipment we used a conventional PC running a non-real-time operating system. Such a system has a low cost; it is relatively easy and quick to build and run a valid and useful HIL simulation for embedded system with low I/O rates with a less complicated simulation environment.

The access of the external devices by the embedded processor is intercepted and forwarded to the simulator running on the PC to make the programming of the embedded processor as realistic as possible. To achieve this, we developed a library of device drivers instead of accessing the device registers directly. The drivers then can be configured to either access the devices on the physical embedded processor or to communicate with the simulator running on the host computer. Other approaches such as running simple device simulators directly on the embedded processor or using the debug mode on the embedded processor can be used to intercept the direct access to the device registers. However, these solutions are not easily portable to other embedded processors and limit the system to a specific application.

The easiest way to communicate with the host computer is through a serial port. Each access to the device is encapsulated in a packet mentioning the address of the device and operation to be performed on that device. This packet is then sent over the serial line to the host computer, which forwards it to the pertinent device simulator. Similarly, if an interrupt is generated by any device simulator, the DSP control engine on the host computer sends a packet over the serial line mentioning the source of interrupt. The interrupt handler on the DSP then decodes the packet and forwards the interrupt to the user defined interrupt handler.

To facilitate the use of the virtual testbed in a laboratory environment, we provide a mechanism for controlling the device drivers while the user code is running on the DSP. This is done by allowing the user to switch between accessing actual hardware devices and forwarding requests to the simulators. This will help debug the device simulators by running the simulation in parallel with external hardware devices and comparing the result of the simulation with the results obtained from the hardware devices. The processor abstraction of the simulator is simply an abstraction of the registers and other devices that are present on the embedded processor. This abstraction provides easy access to various devices and their data on the embedded processor for the simulation core. The updates of the values sent to processor abstraction from the EVM are transparent to the simulation core. Figure 14 shows the hardware and software layers on the embedded processor.

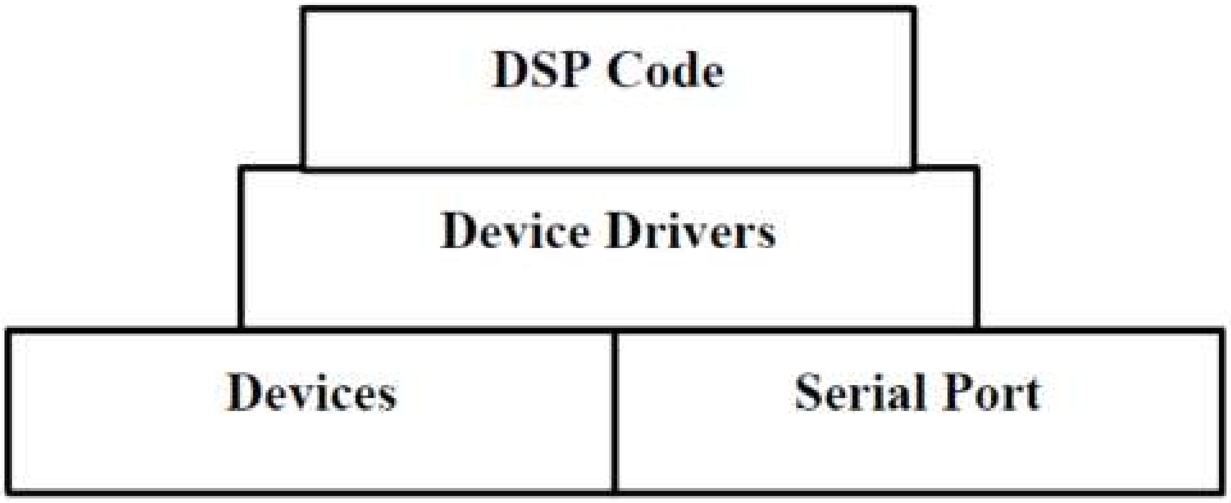


Figure 14: Hardware and Software Layers on the Embedded Processor (modified from [9])

Figure 15 shows the detailed architecture of a virtual testbed. Underlying Windows serial port drivers are not shown to make the diagram less complex. The simulator is connected to the EVM using a serial cable. We can disconnect the simulator and the EVM; in this case the EVM can only be connected to the physical external devices. This system can no longer be called HIL simulation, since we do not have a computer connected to the system. If we disconnect external devices from the EVM as well, then the system resembles a rapid prototyping system. If we have both simulator and the external devices connected to the system then the system can be used to monitor the output on the simulator. This will help to understand the system behavior and can be used for study purposes.

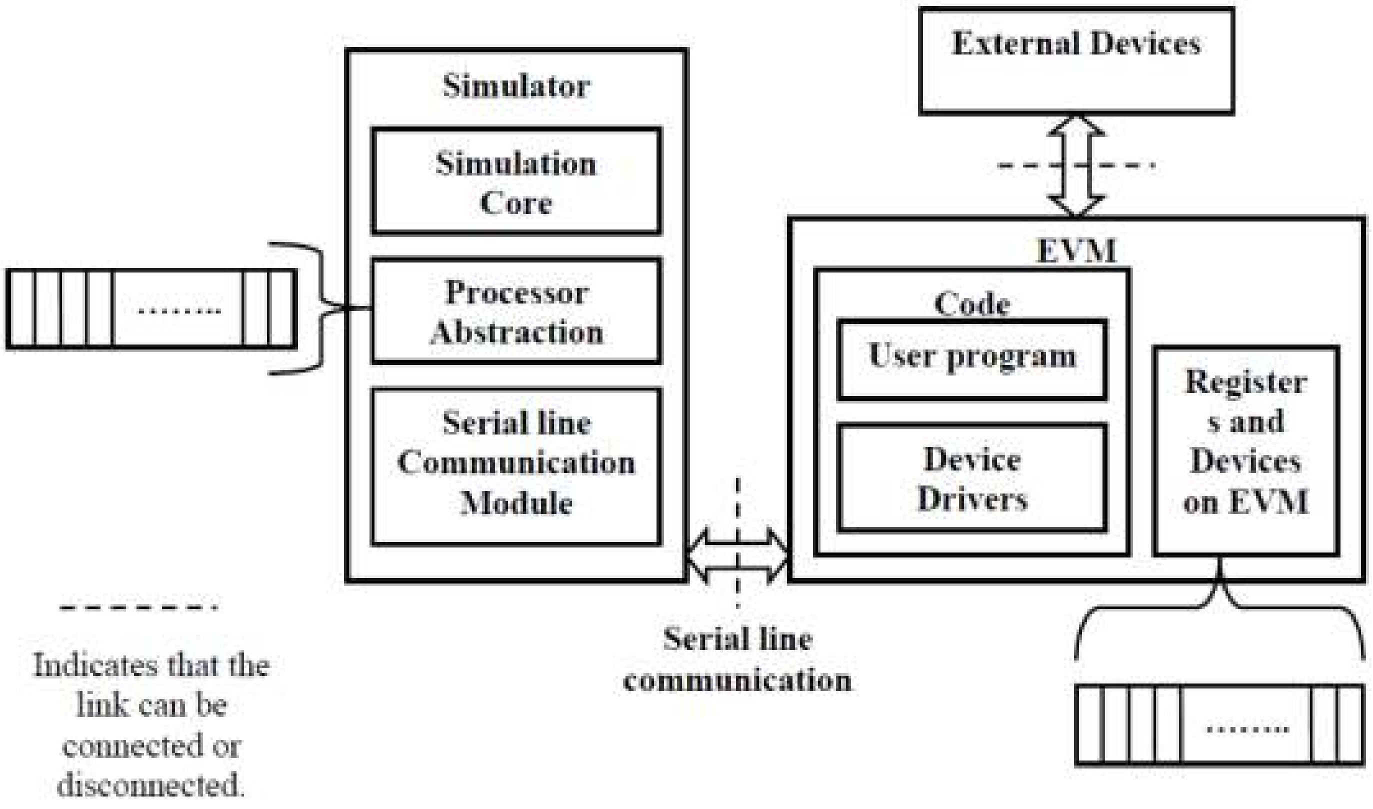

Simulator
Simulation Core
Processor Abstraction
Serial line Communication Module
External Devices
EVM
Code
User program
Device Drivers
Register s and Devices on EVM
Serial line communication
Indicates that the link can be connected or disconnected.

Figure 15: Detailed Architecture of Virtual Testbed [6]

3.4 System Behavior

This section explains the general behavior of the system. As explained earlier the simulator has two components, an EVM and a host PC that runs the simulator. The simulator runs the code that simulates the traffic, displays the traffic light, simulates the street sensors and sends messages to the EVM over the serial line based on the events. The simulator simulates two streets named Main Street and Side Street. The traffic on Main Street is simulated by entering a numeric value into a textbox that is treated as the number of vehicles per minute, for Main Street. The traffic for Side Street traffic is generated by pressing one of the Side Street buttons. The simulator listens over the serial line using a .NET component. We used Code Composer Studio to write the code in C, to compile the code and to load the code into the embedded processor. The EVM has a built-in IEEE JTAG emulation mini-USB connector, which is used to download the compiled program into the embedded processor and to run, halt and debug the program.

Once the program in the embedded processor is running, it transmits the packets over the serial line. The format of the packet and the protocol of this communication are discussed in detail in the next section. Upon receiving a packet over the serial line, the simulator decodes the packet and updates the traffic light as per the data in the packet. We have developed different cases to test the system. The test case "TrafficLightCycler" keeps updating the traffic lights after a pre-programed time. The test case "TestNetLink", is used to check if the communication between the EVM and simulator is working. The other test cases are explained in detail in Chapter 6, which helps measure the performance of the system under different conditions. Figure 16 shows the interface of the traffic light simulator.

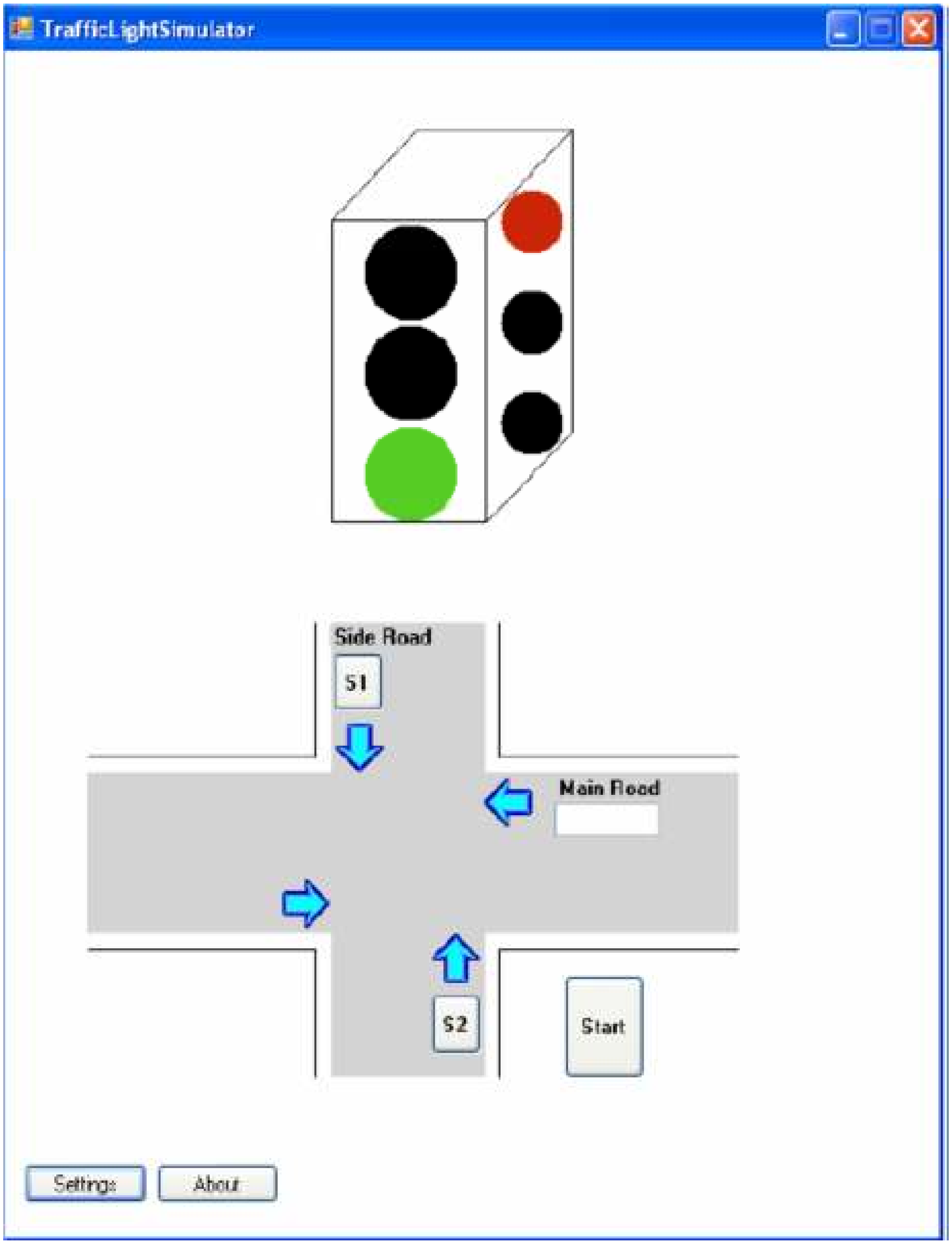


Figure 16: The Interface of the Traffic Light Simulator

CHAPTER 4. IMPLEMENTATION

4.1 Porting Software onto a 32-bit DSP

In the early phase of development we were using a 16-bit DSP, TMS320LF2407A. To facilitate more features we changed to a more advanced 32-bit DSP, TMS320F28335. One of the main tasks for us was to make the existing software compatible with the new DSP. Since the new DSP is 32-bit, most of the registers on the DSP are also 32-bit; therefore, their structure is very different from the previous DSP. For example, the new DSP has separate registers to set the direction of the device and to read/write to the device, whereas, the previous DSP had a single register to set the direction and to read/write to the device. Hence, we have to rewrite the device drivers, taking the 32-bit structure of the registers in to account. Also, the hardware addresses and the names of the device registers were different. For example, in the previous DSP the multiplexer register for port A is named as MCRA, whereas in the present DSP it is divided into two registers, each 16-bit and named GPAMUX1 and GPAMUX2. Also, the hardware address for the port A data and direction register in the previous DSP is 7098(H) whereas in the present DSP, ths hardware address is split into 6F8A for direction and 6FC0 (H) for data.

The program is arranged in modular form to select the task to be performed. This is achieved by developing different test cases. To make the program more users friendly, different tasks are defined as variables, for example the test case to measure the round trip time is named RTT, which is an abbreviation for round trip time. We also developed Get and Set functions to receive and send data over the serial line. A function is developed to set the mode of operation for all registers. The timers and external interrupts on the embedded processor were initialized to facilitate the measurements. All of these are explained in detail in section 4.3.

4.2 Device Drivers

a. Serial Communication Interface Drivers

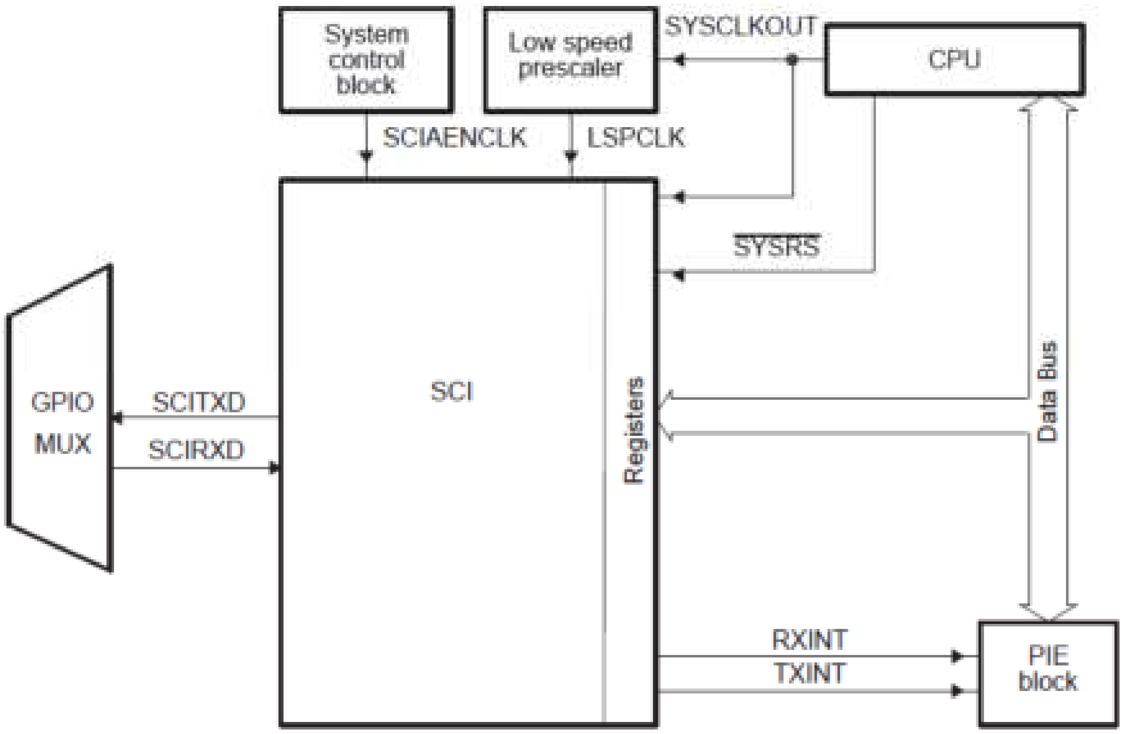


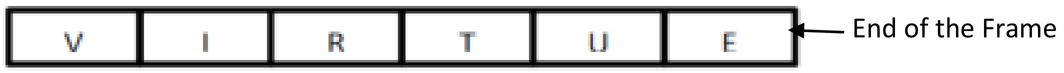
Figure 17: SCI CPU Interface [32]

The above figure shows the interface between the CPU and the serial device. The serial communication interface (SCI) is a two-wire asynchronous serial port commonly known as UART which supports the digital communication between the CPU and other asynchronous peripherals. The baud rate is programmable to 64K different speeds through a 16-bit baud-rate-select register. The baud rate is determined by a low-speed peripheral clock (LSPCLK) and the baud-select registers. The SCI asynchronous baud rate is given by the formula

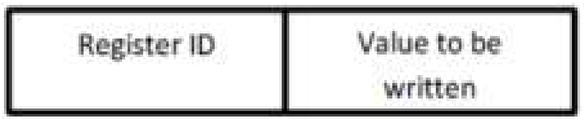
$$SCI\ Asynchronous\ Baud = \frac{LSPCLK}{(Baud\ Rate + 1)\ X\ 8}$$

Where the LSPCLK can be set equal to the system clock or can be pre-scaled with respect to the system clock. The maximum pre-scale value is 14. If we use a LSPCLK which has been made the same as the system clock, then the maximum baud rate we can get is 18.7 Mbps and the minimum baud rate is 20 bps. We are using 115 Kbps since this is the maximum that the serial device on the host computer can handle. However, we were able to test the baud rate of 921.6 Kbps with the help of a USB2SERIAL converter. We used HyperTeminal to receive the data transmitted by the DSP over the serial line. Each frame has one STOP bit and one START bit.

We developed two functions to handle the communication over the serial line namely *Get* and *Set*. As the names suggest the get function is used to receive the packet over the serial line and the set function is used to transmit data over the serial line. We have developed two types of packets that take different measurements to test the performance of the system. For example, to test round trip time (RTT) for different numbers of bytes, we send the word 'VIRTUAL', over the serial line and replace a character with 'E' that makes the desired number of bytes. Let's say we want to measure the RTT for 6 bytes then we send 'VIRTU' and 'E' over the serial line as shown in figure 18a. 'E' marks the end of the packet to stop the timer. To measure the simulation speed we send a packet containing the register identifier (ID) and a value to be written to the register as shown in figure 18b.



a



b

Figure 18: Packet Formats

The following is how we are assigning the ID's to the registers. The general purpose input/output registers (GPIO) are located starting from the address 6F80 (H) to 6FE8 (H); however, not all registers are adjacent. For example, the hardware address for GPIO A Pull UP Disable Register (GPAPUD) is 0x6F8C and the next register, GPIO B Control Register (GPBCTRL) has an address 0x6F90, that is the three addresses 0x6F8D, 0x6F8E and 0x6F8F are not used. Also, some registers are 32-bit and some are 16-bit; hence, we are classifying the register based on their size. We are using the distance of the register from the first GPIO register and its size to calculate the ID of the register. For example, the ID for the GPBCTRL register will be given by subtracting the address of the first GPIO register from the address of GPBCTRL register (0x6F90 – 0x6F80 = 0x10 i.e. decimal 16). Now, as these registers are 32-bit, we divide the subtraction by two to get the ID for the register GPBCTRL (16/2 = 8). If the register is 16-bit then the ID is simply its distance from first GPIO register, for example, the ID of XINT2 GPIO Input Select Register (GPIOXINT1SEL) is 97 (0x6FE1 - 0x6F80 = 0x61 that is decimal 97).

Upon receiving the packet from the serial line, action is taken depending upon the packet. If the packet is to update a register then the register ID and value are forwarded to a function called `set` to calculate the physical address of the register, if the mode of operation is hardware mode then the contents are updated. As shown in figure 19, upon receiving the register and ID and value, it first checks if the mode of operation is hardware or simulation mode. If the mode of operation is hardware mode, the size of the register is checked. If the size of the register is 32-bit then the register ID is multiplied by two and then ORed with 0x6F00 to get the memory address. A pointer is then assigned to this address and the value is written to it. Let's say the register ID is 32 then the hardware ID for this register will be 6FC0 ((32*2) + 128 = 192 = 0xC0 ORed 0x6F00 = 6FC0).

```
 void set(register_code reg, long value)
{
     int memory_add;
     int *memory_ptr;
     sim_mode reg_mode = mode(reg);
     if (reg_mode & HW_MODE)
     {
         if(reg_size == 32)
         {
            reg_size = 0;
            memory_add = (2 * reg) + 128;
         }
         else
         {
            reg_size = 0;
            memory_add = reg + 128;
         }
         memory_add = memory_add | 0x6F00;
         memory_ptr = (int*)memory_add;
         *memory_ptr = value;
     }
     if (reg_mode & SIM_MODE)
     {
         sim_write(reg, value);
     }

  }
```

Figure 19 Sample code to calculate the address for a register

b. Determining Mode of Operation for a Register

We developed two functions, `set_mode` and `sim_mode`, to set and determine the mode of operation for a register. We are using an integer array to store the mode of operation for each register. The integer variable is 16-bit hence; each element of the array will hold the mode of operation for eight registers if we assign two bits, one bit each for the two modes of operation, to each registers. The first bit for a register is set if the mode of operation for that register is simulation mode, similarly the second bit is set if the mode of operation is hardware mode. Figure 21 shows the flowchart for the `sim_mode` function. The register ID is passed to this function as an argument and the function then returns the mode of operation. The variable `index` points to the array element which has the mode of operation stored for the register. The variable `offset`, points to the bits of the array element that stores the mode of operation. For example, if the register ID is 40 then the `index` will be four and the `offset` will be eight, as the array index starts from zero. The variable `mask` is also a 16-bit integer; we are using this variable to AND with the array element that stores the mode of operation. Figure 20 shows a sample element of the mode array. Figure 22 shows a piece of program demonstrating the function `sim_mode`.

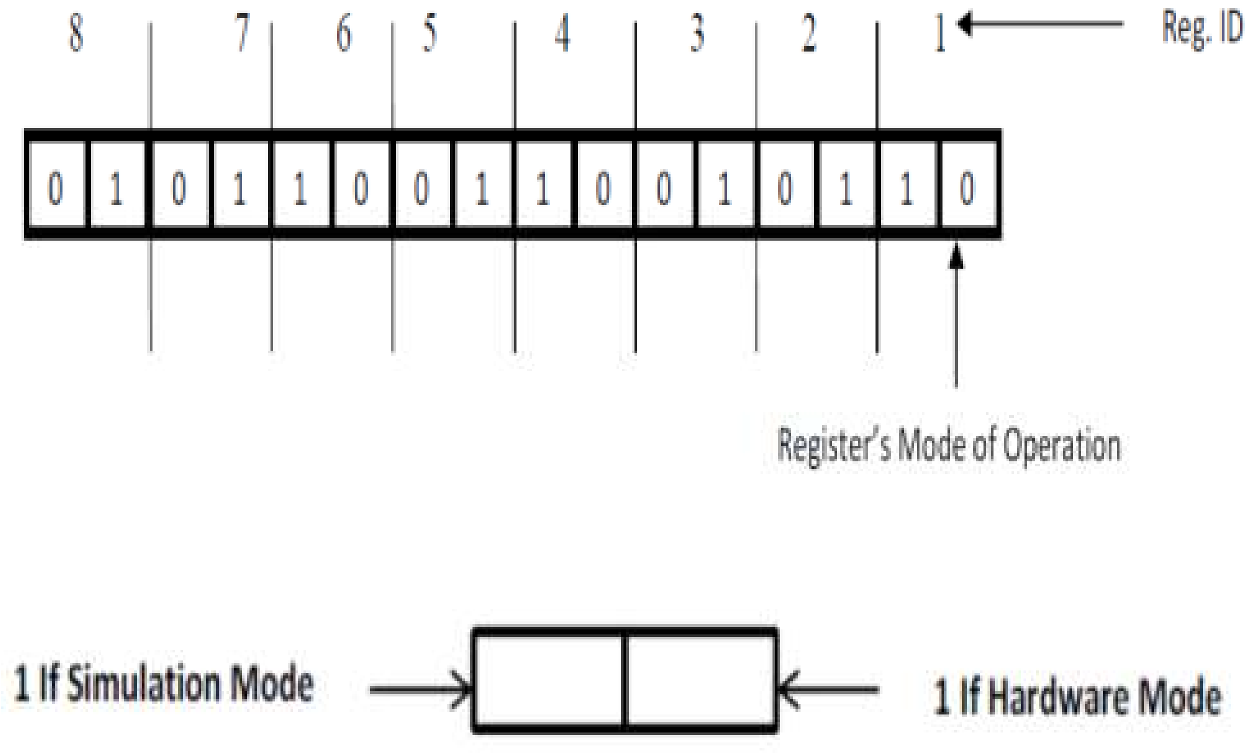


Figure 20: Sample Mode Integer

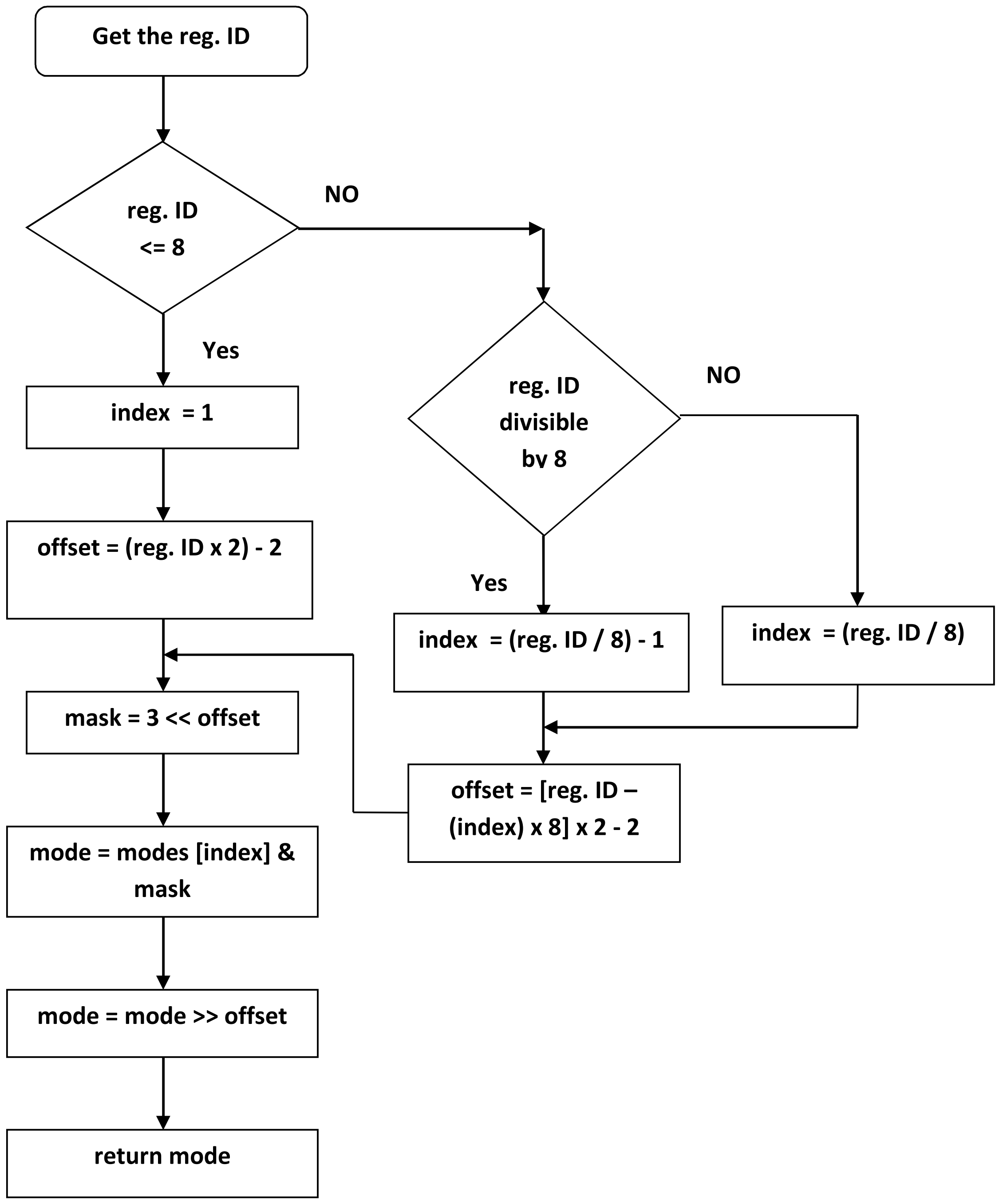


Figure 21 Flowchart for “sim_mode” Function

```
sim_mode mode(register_code reg)
{
    int index = 0;
    int offset = 0;
    int mode = 0;
    int mask = 0, result = 0;
    if(reg <= 8)
    {
       index = 1;
       offset = reg;
       offset = (offset*2) - 2;
       mask = (3<<(offset);
      mode = (modes[index] & mask) >> ((offset);
      return (sim_mode)mode;
    }
    else
    {
      result = reg & 0x07;
      if(result == 0)//reg = 40
      {
        index = reg / 8; // index = 5
        index = index - 1; // index = 4
        offset = index * 8; // offset = 32
        offset = reg - offset; // offset = 40 - 32
        offset = (offset*2) - 2;
        mask = (3<<(offset);
        mode = (modes[index] & mask) >> offset;
        return (sim_mode)mode;
     }

     else
     {
        index = reg / 8;
        offset = index * 8;
        offset = reg - offset;
        mask = (3<<offset);
        mode = (modes[index+1] & mask)>> offset;
        return (sim_mode)mode;
     }
   }
}
```
Figure 22: Program showing the “sim_mode” Function

c. Timer Drivers

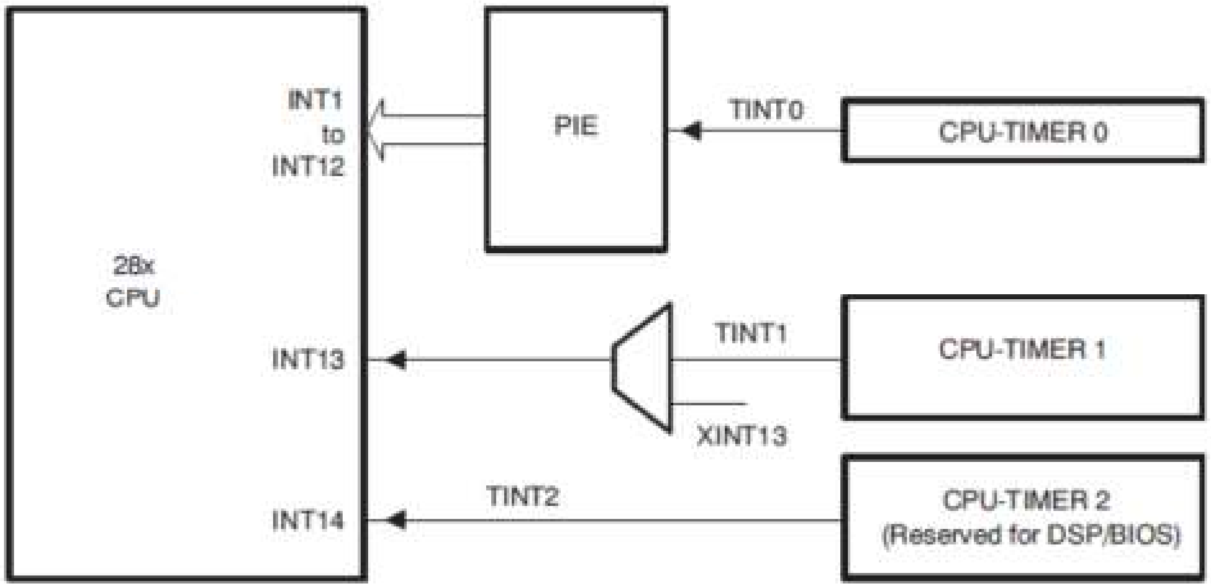


Figure 23: CPU-Timer Interrupt Signals and Output Signal [33]

As stated in Chapter 3, the embedded processor TMS320F28335 has three 32-bit timers. Figure 23 shows the block diagram for CPU-Timer interrupt signals and output Signal. We are using these timers to measure the performance of the system. We are using the timer count and the timer interrupt to calculate the elapsed time. Figure 21 shows different CPU-timer interrupt signals. The 32-bit period register has the period value which is loaded to the counter register when the timer starts. The counter decrements by one after every timer divides down plus one timer clock source cycle and an interrupt is generated once the timer count decrements to zero. We have configured the timer to get the maximum granularity; to achieve this we are setting the pre-scaler to one. This allows the timer clock frequency to be the same as that of the system clock frequency, which is 150 MHz in our case. With this clock frequency we are able to get a clock tick of 6.67 ns. The maximum time period we can get with this clock configuration is approximately 29 sec.

Figure 24 shows the C code for timer initialization. This function initializes the address pointers to their respective timer register; it also loads the timers period register with the maximum possible value and sets the pre-scaler value to one. It also sets the time stop status bit to one to make sure that the timer is not running and reloads the counter register with the period value. It also clears the interrupt count bit to reset the interrupt counter. The interrupt count is used to calculate the time if the time required by the process is more than the maximum possible time with our present timer configuration. Figure 25 shows the interrupt service routine for the timer. In this routine we are incrementing the variable cpuTimer0.InterruptCount upon every interrupt generated by the timer.

```
void InitCpuTimers(void)

{

    CpuTimer0.RegsAddr = &CpuTimer0Regs;

    CpuTimer0Regs.PRD.all  = 0xFFFFFFFF;

    CpuTimer0Regs.TPR.all  = 0;

    CpuTimer0Regs.TPRH.all = 0;

    CpuTimer0Regs.TCR.bit.TSS = 1;

    CpuTimer0Regs.TCR.bit.TRB = 1;

    CpuTimer0.InterruptCount = 0;

}
```

Figure 24: C code to Initialize the Timer

```
 interrupt void cpu_timer0_isr(void)
{
   CpuTimer0.InterruptCount++;
   PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
}
```

Figure 25: C code for Timer Interrupt Service Routine

The timer interrupt routine measures the number of the timer overflows and also acknowledges the interrupt to allow receiving more interrupts. The number of interrupt counts is used to calculate the time needed to send data over the serial line. In the interrupt servicing routine, we start the timer to measure the time taken by the DSP to react to the external event. The time is then sent to the simulator over the serial link, which then writes it to a text file.

4.3 Simulator

As previously stated, the simulator runs on the host PC and has software abstractions of hardware devices [6]. The simulator for the virtual testbed has two components as shown in Figure 26, the simulation core and a Microsoft .Net framework component. The Microsoft .Net framework has three sub-components, processor abstraction, protocol encoder/decoder and the serial port communication component and the simulation core contains the control logic for the simulator. A number of specialized programming languages and environments along with their respective libraries, has been proposed over the years for simulation development. We have developed the simulator using Microsoft Visual #C for .Net [14], [22]. Microsoft .Net framework is a platform for the development of various applications on Windows operating systems. The .Net framework provides a comprehensive and consistent programming model for building applications along with seamless and secure communication. The class library of .Net framework is called the Framework Class Library (FCL) which is nothing but a collection of a number of classes and most of the classes are wrappers for Windows APIs.

Liu [15] used Windows' file abstraction to read and write from the serial port. However, its limitation was identified by [6]. To overcome this Medavarapu [6] used a .Net SerialPort component that is present in the System.IO.Ports namespace, which contains classes for controlling serial ports. This SerialPort component provides a framework for synchronous and event-driven I/O, access to pin and break states and access to serial driver properties [23]. An event-driven framework for the processor abstraction is developed by [6] with these properties of the Serial Port component. This event-driven handler is invoked when the value of a register in the processor abstraction changes.

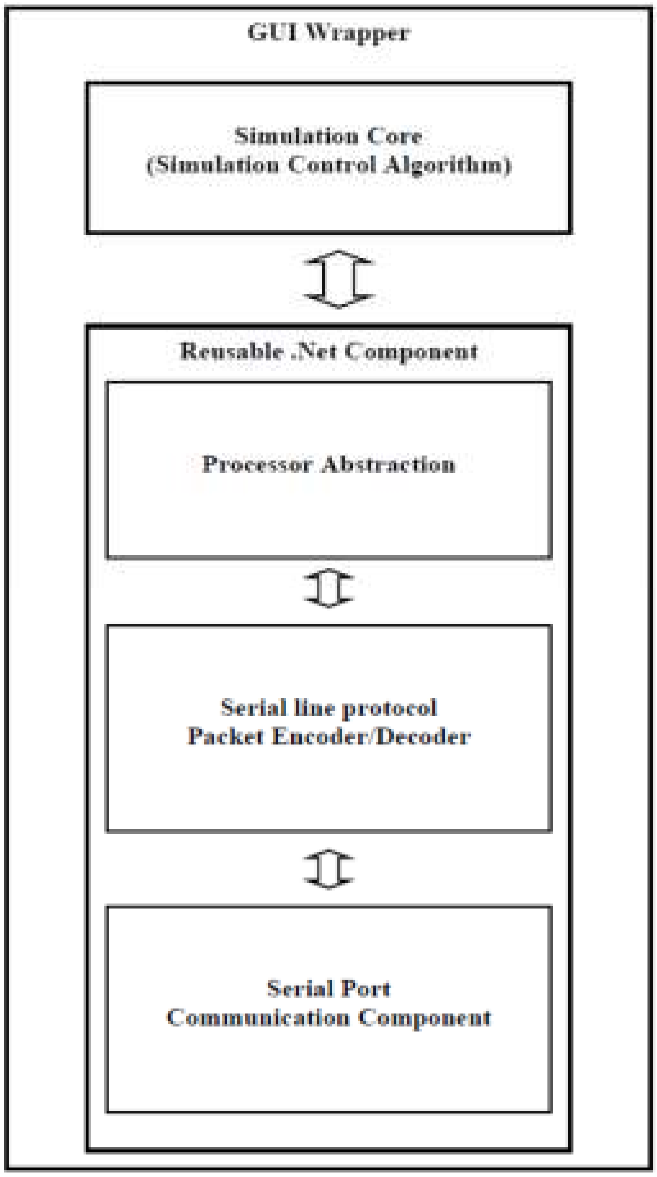


Figure 26: Simulation Components [6]

The synchronization mechanisms of the simulation device are used for accurate timing, the simulation frame is one such mechanism. The simulation frame can be defined as the interval of time of one pass from reading the input, running the control logic, to sending the output. The simulation frame of HIL simulation is shown in figure 27. It is better if the simulation frame is short to have accurate simulation, but it should be long enough to tolerate the worst-case time needed to complete all necessary operations in the simulation. A shorter simulation frame can be achieved by simplifying the simulation calculations. As the simulation frame is lengthened the simulation accuracy worsens.

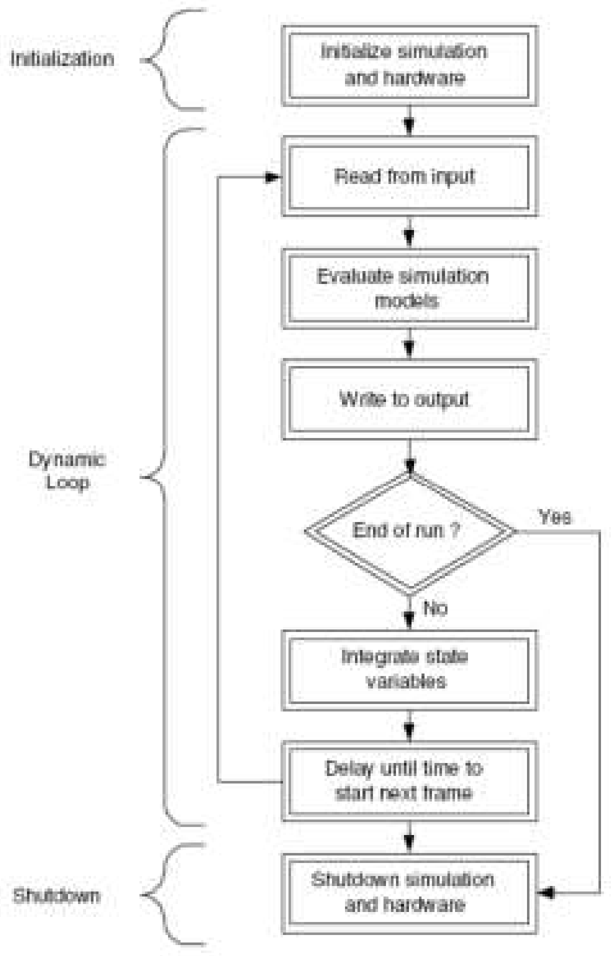

Initialization
Initialize simulation and hardware
Read from input
Evaluate simulation models
Write to output
Dynamic Loop
End of run ?
Yes
No
Integrate state variables
Delay until time to start next frame
Shutdown
Shutdown simulation and hardware

Figure 27: Program Flow of a HIL Simulation Frame [15]

We further developed the simulator to add more functionality and to make it more user friendly. For example, we improved the logic that updates the light making it easier to understand. Also, we implemented an `Update` function to update the registers with the new value. This function takes the register ID and value as arguments, which are then passed from the decode function. We invoked the system timer to measure the performance of the simulator. The System.Timers namespace provides the timer component, which allows the raising of an event on a specified interval. We also added the logic to facilitate different measurements. Figure 28 shows a piece of a program in C# demonstrating the calculation of time and the writing of that time to a text file. We are using the `StreamWriter` class within the System.IO namespace to write the time to the specified text file. We also developed a method `SendByte`, which sends the desired number of bytes over the serial line. This function is used to check the round trip time of the system. Figure 29 shows the method for updating the traffic lights. This function is explained in section 4.2.a. Figure 30 shows the code in C# for the method “SendByte”.

```
string path = @"c:\Shri\Shri_MS\thesis\test2.txt";
StreamWriter sw = File.AppendText(path);
double ts = stopwatch.ElapsedMilliseconds;
long ticks = stopwatch.ElapsedTicks;
double ns = 1000000000 * (double)ticks / Stopwatch.Frequency;
MessageBox.Show("ns.RunTime " + ns);
stopwatch.Reset();
sw.WriteLine(ns);
sw.Close();
```

Figure 28: C# Code for Time Calculation

```
 private void dsp_RegisterUpdatedEvent(DSPEventArgs e)
{
       int Recvdata = (dsp.GetValue(Registers.GPBDAT));
       byte result = (byte) (Recvdata & 0xff);
       clearLights();
       if (contains(result, IOPortBit.MAIN_RED))
           setLight(Street.MAIN, TrafficLightColor.RED);
       if(contains(result, IOPortBit.MAIN_GREEN))
           setLight(Street.MAIN, TrafficLightColor.GREEN);
       if (contains(result, IOPortBit.MAIN_YELLOW))
           setLight(Street.MAIN, TrafficLightColor.YELLOW);
       if (contains(result, IOPortBit.SIDE_RED))
           setLight(Street.SIDE, TrafficLightColor.RED);
       if (contains(result, IOPortBit.SIDE_GREEN))
           setLight(Street.SIDE, TrafficLightColor.GREEN);
       if (contains(result, IOPortBit.SIDE_YELLOW))
           setLight(Street.SIDE, TrafficLightColor.YELLOW);
       // force a repaint
       Invalidate();
   }
```

Figure 29: C# Code for Register Update

```
 public void SendByte(int No_Bytes)
{
   byte[] bytearry1 = new byte[10];
   bytearry1[0] = (byte)'V';
   bytearry1[1] = (byte)'I';
   bytearry1[2] = (byte)'R';
   bytearry1[3] = (byte)'T';
   bytearry1[4] = (byte)'U';
   bytearry1[5] = (byte)'A';
   bytearry1[6] = (byte)'L';
   bytearry1[No_Bytes-1] = (byte)'E';
   int i = 0;
   stopwatch.Start();
   for (i = 0; i <= No_Bytes-1; i++)
  {
     try
     {
         serialPort.Write(bytearry1, i, 1);
     }
     catch (Exception ex)
     {
         MessageBox.Show(ex.ToString());
         throw;
     }
 }
}
```

Figure 30: C# Code of a Method for Sending Bytes over Serial Line

CHAPTER 5. MEASUREMENTS

5.1 Different Test Cases

a. Round-trip Time

We are measuring the round trip time for one, two, four and six bytes. To measure the round trip time we are making use of the system timer. The desired number of bytes is sent to the embedded processor and the timer is started. The embedded processor then responds by acknowledging the receipt of the packet. Upon receiving this acknowledgement from the embedded processor the timer is stopped and the time is written to a text file. The time granularity is in nanoseconds.

b. Simulation Mode Measurements

To measure the performance of the system in simulation mode, we are making use of system timers, which are invoked by the simulator. The time is measured in nanoseconds. In this case we are pressing a switch in the simulator that simulates the street sensor in one of the streets. Then in the simulator we decode the location of the switch, start the timer and send the packet over the serial line to the embedded processor. As stated in Chapter 4, the packet contains the register ID and the value to be written to the register. On the EVM we decode the packet updates the register and send the next light sequence if it is time for the light to change. Once the packet is received over the serial line on the simulator side, the packet is decoded and the lights are updated if required and the timer is stopped. The time is then written to a text file for reference. Figure 31 shows the flowchart for the simulation measurements.

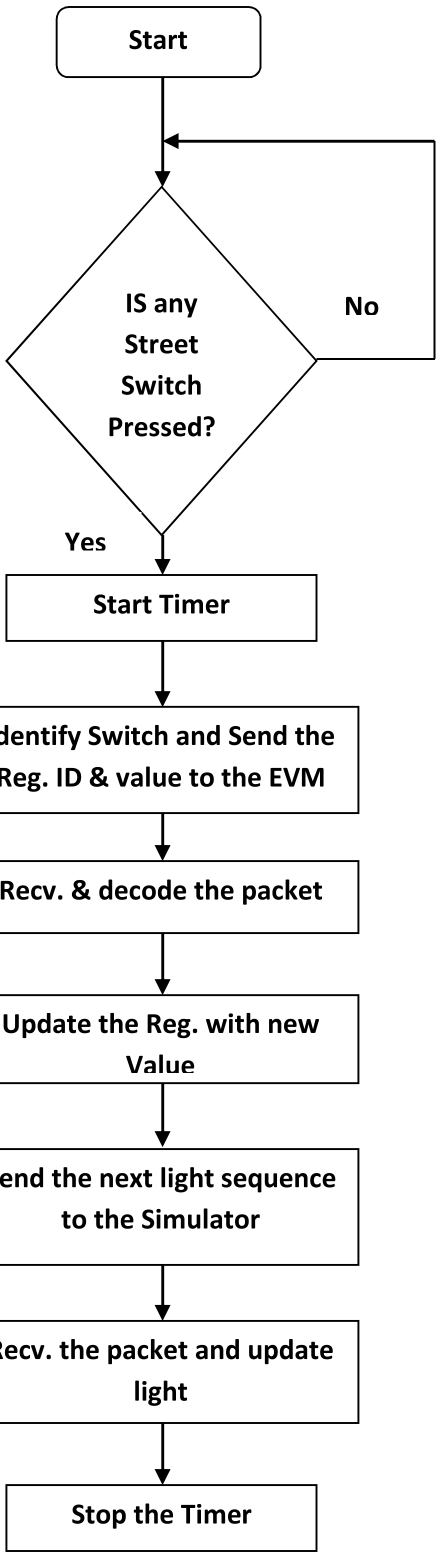

Start
IS any Street Switch Pressed?
No
Yes
Start Timer
Identify Switch and Send the Reg. ID & value to the EVM
Recv. & decode the packet
Update the Reg. with new Value
Send the next light sequence to the Simulator
Recv. the packet and update light
Stop the Timer

Figure 31: Flowchart for Simulation Measurements

c. Hardware mode Measurements

Here we are using the timer to check the performance of a real embedded system that has hardware attached to it. Our embedded system simply a traffic light controller. We have connected an external switch to the embedded processer to emulate the street sensors of a real traffic light. When the switch is pressed, to indicate the passing of a vehicle, an interrupt is generated and the timer is started in the interrupt service routine. We update the number of vehicles on either the main street or on the side street based on the source of interrupt. Then we read the parallel port to check the status of the lights and decide whether or not to update the light. If we need to update the lights, we send the next sequence of lights to the parallel port and stop the timer. Then the time taken by the process is calculated using the "Cal_Send_Time" function and transmitted over the serial line to the simulator. On the simulator the time value received is written to a text file. Figure 32 shows the flowchart for hardware mode measurements.

d. Throughput

To measure the throughput of the system we are sending a thousand bytes from the simulator to the DSP. Upon transmitting the packet to the embedded processor the system timer that is invoked in the simulator is started. The embedded processor acknowledges the receipt of the packet by sending a packet containing an "A". The timer is stopped upon receipt of this timer and the time is written to a text file.

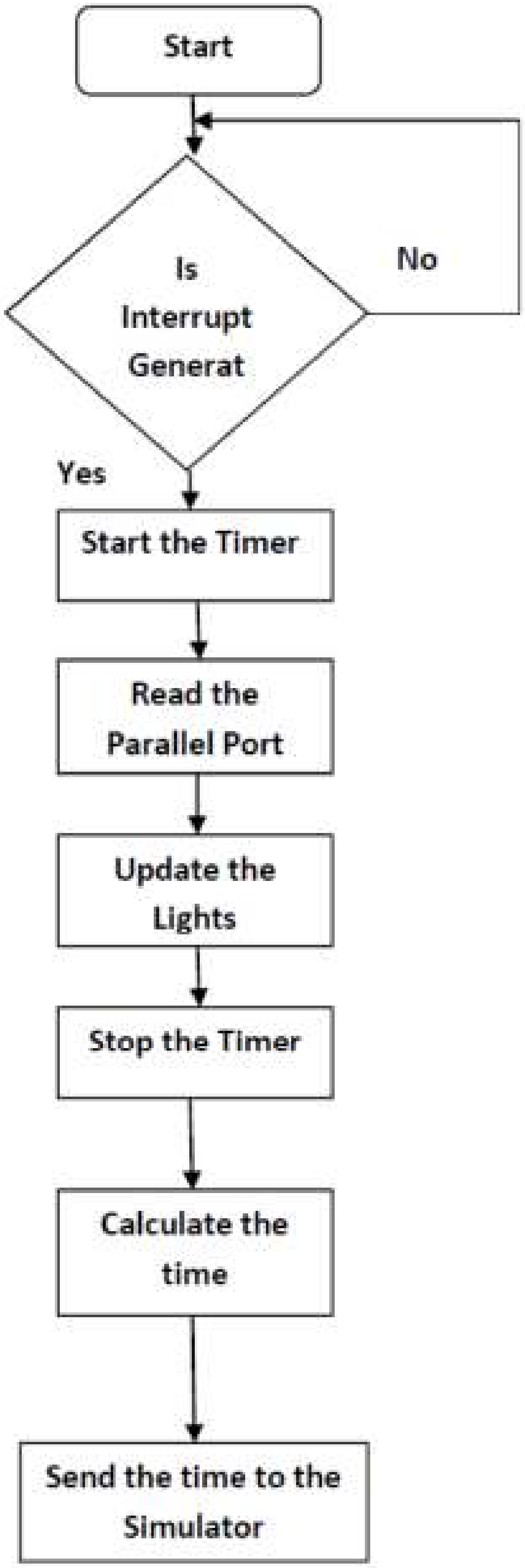


Figure 32: Flowchart for Hardware Mode Measurements

5.2 Measurements

The measurements are a very important part of any system to verify its behavior. The measurements also help to identify the areas where the system can be used effectively. The measurements also help the engineer to judge the usefulness of the system in his or her application. Measurements also highlight the drawbacks of the system and the areas of improvements. It is very important for any measurements to be as realistic as possible to judge the performance of the system under consideration, and when the measurement involves a non-real time operating system we need to take care to minimize the effect of the operating system on the measurements**.**

Since we used a non-real-time operating system, most of the measurements were run 100 times after warming up the cache to minimize the effect of the operating system. Once we had the readings we calculated the average of the readings. From the average we calculated the standard deviation for the readings. The standard deviation helps to identify the variation of readings from the average. To make the readings more pertinent we discarded the readings four standard deviations away from the mean. This process of discarding outliers was repeated twice to eliminate any influence of operating system interrupts. The readings were taken with a serial cable, which connected the embedded system and the serial port of the host PC and with a USB2SERIAL converter. The baud rate for the communication was 115 Kbps since this is the maximum possible baud rate with the PC serial port. As mentioned earlier in Chapter 4 each frame consists of a STOP and START bit and no parity bit. Table 1 and 2 shows the reading for performance of the system with different sizes of data flow over the serial line.

Table 1: Measurements with Serial Cable (115Kbps)

File	# Valid Readings	Avg. Run Time (ms)	Std. Dev. (ms)	Min. (ms)	Max. (ms)	Error (ms)
RTTByte1	95	1.13	0.02	1.068	1.206	0.004
RTTByte2	98	1.36	0.39	1.015	2.852	0.78
RTTByte4	98	1.68	0.42	1.406	3.455	0.08
RTTByte6	95	1.91	0.83	1.740	2.212	0.01
Throughput	99	97.1	1.51	95.291	102.355	0.30

Table 2: Measurements with USB-to-Serial Converter (115Kbps)

File	# Valid Readings	Avg. Run Time (ms)	Std. Dev. (ms)	Min. (ms)	Max. (ms)	Error (ms)
RTTByte1	98	1.0	0.63	8.183	1.272	0.012
RTTByte2	95	1.4	0.45	1.147	3.204	0.092
RTTByte4	96	2.04	0.10	1.582	2.364	0.021
RTTByte6	98	2.68	0.14	2.155	3.031	0.029
Throughput	100	186.38	21.73	139.269	242.846	4.318

The column “File” contains the number of bytes send over the serial line, for example, the “RTTByte1”, means the reading is for a round-trip time of one byte. The last entry “Throughput”, is the round-trip time for a thousand bytes. The column “# Valid Readings” shows the actual number of readings used to calculate the average run time out of a given hundred readings after discarding readings that were four standard deviations from the mean. In case of “RTTByte3” the number of valid readings is ninety eight this means we had two readings, which were more than four standard deviations away from the mean. The “Min” column shows the minimum value among the readings and the “Max” column shows the maximum value among the readings. The column “Error” shows the error bar for each reading.

It can be seen in the table that the average run time increases steadily as the number of transmitted bytes increases. This holds with the fact that a greater number of bytes takes more time to be transmitted and also proves the validity of the system. Also, if we compare the readings between the serial cable and the USB2SERIAL converter, we can see that the time taken by the latter is more. This can be due to the hardware in the USB2SERIAL converter. However, the USB2SERIAL converter can help us test the system with higher baud rates. As stated in Chapter 4 the maximum possible baud rate with the DSP is 18.7 Mbps, which is slightly more than USB1 speed, and cannot be tested with conventional serial ports. From Table 1 and Table 2, it takes approximately ninety eight milli-seconds and one hundred and eighty seven milli-seconds to send one thousand bytes (Throughput) at the speed of 115 Kbps with a serial cable and with a USB2SERIAL converter. Whereas the expected time is,

$$\text{Expected Time} = \frac{1000}{115000} \quad X \quad 1000 = 86.97\sim 87 \text{ ms}$$

$$\text{Actual Time To Expected time Ration For USB2SERIAL} = \frac{187}{87} = 2.14 \quad \text{- - - } 1$$

Hence, the **effective baud rate** for USB2SERIAL is,

$$\frac{1000}{187 \text{ ms}} \quad X \quad 1000 = \boldsymbol{\mathit{5.34\ Kpbs}}$$

Similarly,

$$\text{Actual Time to Expected time Ration For Serial Cable} = \frac{97}{87} = 1.11$$

Hence, the **effective baud rate** for serial cable is,

$$\frac{1000}{97 \text{ ms}} \quad X \quad 1000 = \boldsymbol{\mathit{10.30\ Kpbs}}$$

If the embedded processor is set to the maximum baud rate, 18.7 Mbps, which is approximately 19 Mbps, the average time for the Throughput for USB2SERIAL will be,

$$\frac{1000}{19\text{ Mbps}} \times 1000 = 52.63 \sim \boldsymbol{\mathit{52\ ms}}$$

From 1, Expected time will be,

$$52\text{ ms} \times 2.14 = 111.28 \sim \boldsymbol{\mathit{111\ ms}}$$

The **effective baud rate** for USB2SERIAL is,

$$\frac{1000}{111\text{ ms}} \times 1000 = \boldsymbol{\mathit{9\ Mbps}}$$

We also took the measurements to test the performance of the system under simulation and hardware mode. These readings were taken with a serial cable and a USB2SERIAL converter to see the effect of the converter. Studying the effect of the converter is useful in judging the application domain of the system. All the fields in the table are similar to those in Tables 1 and 2. Here, it is also evident that the converter takes some time for the conversion and hence takes more time for receiving the data. We can also note that the time taken in hardware mode is less than the time taken in the simulation mode. This again holds true as the simulation is not running on a real-time operating system and has several factors affecting the performance of the system. As shown below, Table 4 shows the reading taken with serial cable and Table 5 shows the readings taken with the USB2SERIAL converter.

Table 4: Measurements with Serial Cable (115Kbps)

File	# Valid Readings	Avg. Run Time (ms)	Std. Dev. (ms)	Min. (ns)	Max. (ns)	Error (ns)
Simulation Mode	99	0.643	0.115	0.318	1.00	0.023
Hardware Mode	100	0.039	0.109	0.257	0.50	0.015

Table 5: Measurements with USB-to-Serial Converter (115Kbps)

File	# Valid Readings	Avg. Run Time (ms)	Std. Dev. (ms)	Min. (ms)	Max. (ms)	Error (ms)
Simulation Mode	96	1.09	0.093	0.323	1.40	0.018
Hardware Mode	100	0.04	0.012	0.03	0.06	0.010

CHAPTER 6. CONCLUSION AND FUTURE WORK

A simulation will be helpful if and only if it depicts the real-time behavior of the system under testing, which can be achieved only when the real-time operating system is used for simulation. As stated in Chapter 4, we are using a non-real operating system, Windows, to run the simulator. Hence the simulator is not suited for complex applications. Also, the communication speed is limited by the hardware on the PC to 115 Kbps. Hence, the performance of the system for high baud rates cannot be evaluated with the existing setup. However, the set is capable of running a valid simulation for an embedded system with slow I/O rate and simulation environment that is not very complex [6]. A system can be developed with the existing setup for audio signal processing as the maximum baud rate is more than what is required for audio signal processing. Traffic light simulation, as well as simulating the traffic, can also be implemented with this setup. However, the setup is not capable of simulating the hardware de-bounces for the electrical switches under testing.

- **More tests cases to check the performance:** The reading now involves just four test cases, two to measure the communication speed and two to check system performance in different scenarios. However, further test cases can be developed to test the performance of the system under different circumstances. This will help to understand the characteristics of the system and we can classify the application domain where this system can be helpful.
- **More precise readings:** The reading can be improved by minimizing the load on the operating system. For example, we can use make use of the keyboard to take measurements instead of the mouse and tune the performance of the operating system using “Performance Options” settings. Additional options can be explored to optimize the performance of the operating system.

- **Time synchronization:** In the current system the simulator and the embedded system use two separate clocks. It will be helpful if they both run using the same clock. This can be achieved by time synchronization. One approach for this is having the simulator dictate the time to the embedded system. In this approach the DSP will wait for the simulator to finish the simulation. Once the simulation is done the simulator will send a packet to the DSP telling it to respond to the event. This will make sure that the embedded system does not go ahead of the simulator. Another approach can be, using the independent clock for both the embedded system as well as for the simulator. If an interrupt is generated then the embedded system will send a packet to the simulator informing it about the interrupt and asking it if it can respond to it. If the simulator is not ready it will send a wait packet to the embedded system. Once the simulation is done the embedded system will be updated to respond to the interrupt.
- **Simple application to test the limits of the simulator:** A simpler application can be developed to explore the technological limits of the simulator; if the application is simple enough, we can run various tests to examine the technological competency of the simulator. A simple application will be easy to simulate and can be developed in less time. One example is an application on a temperature controller, which involves just a sensor; the processor reads the input from sensor and runs the control logic to determine whether or not to turn on the air-conditioner.
- **Web based interface:** Another interesting application can be to interface the simulator to the web. This can be done by running the simulator in a web browser. Here the hardware can be connected to a remote computer. This will provide remote connectivity to the simulator extending the accessibility to a number of users to test the system remotely. A

prototype was developed by Joel Bonnette and Joseph Hamilton as a part of a class project that can be interfaced with this simulator to check its performance.

- **Timed automaton:** The term timed automaton is used to describe a system specification of an embedded system that helps to simplify the testing and development of embedded systems. Deng [11] proposed the use of a state diagram, which is simpler to use and cleaner than the timed automaton [11]. In the state diagram each edge is labeled with a regular expression instead of a single symbol. Our system can be used to test this approach. A prototype was developed by Deng for a HIL virtual testbed that can be incorporated in our simulator to evaluate the usefulness of the state diagram to specify the specifications of an embedded system.

REFERENCES

[1] A. Agarwal. 2013. “Integrating instruction set simulation into a system level design environment.” *Dept. of Electrical and Computer Engineering: Northeastern University.*

[2] A. Keyhani and A.B. Proca. 1999. “A virtual testbed for instruction and design of permanent magnet machines.” *IEEE Transaction on Power Systems, vol. 14,* 795-801.

[3] A. Nohl, G. Braun, O. Schliebusch, R. Leupers, H. Meyr and A. Hoffmann. 2002. “A universal technique for fast and flexible instruction-set architecture simulation.” New York, NY, USA: ACM, 22-27.

[4] A. A. Reyes, A.P. Narayananaswamy and A.Dogan. 2003. “Simulation-based development of real-time, embedded software for cooperative, autonomous aerial vehicles.” *22nd Digital Avionics Systems Conference. Proceedings.* Vol. 2: 8.A.3-1-11.

[5] C. Helmstetter, V. Joloboff and H. Xiao. 2009. “SimSoc: A full system simulation software for embedded systems.” *International Workshop on Open-source Software for Scientific Computation.* Guiyang, China.

[6] C. Medavarapu. 2007. “An Architecture for Embedded System Sommunication.” M.S. Thesis, *Dept. of Computer Science: Louisiana State University.*

[7] D. Bullock and T. Urbanik. 2000. “Hardware-in-the-loop evaluation of traffic signal systems.” *Tenth International Conference on Road Transport Information and Control.* 177-181.

[8] F. Ghenassia, Ed. 2005. “Transaction-level modeling with systemC. TLM concepts and applications for embedded systems.” Springer.

[9] G. Baumgartner and A. Keyhani. 2000. “A virtual embedded system testbed for instruction and design.” *Digital Signal Processing (DSPS) Fest,Texas Instruments.* Houston, Texas.

[10] Gnu debugger iss. http://www.gnu.org/software/gdb/.

[11] G. Deng. 2007. "Generating embedded system analyzer." M.S. project report, *Dept. of Computer Science, Louisiana State University*.

[12] "Hardware in the Loop Traffic Simulation Final Report." 2005. *Transport Research Centre, University of Florida.*

[13] H.-J. Herpel, M. Glesner, H. Eggert, W. Suss, M. Gorges-Schleuter and W. Jakob. 1995. "Rapid Prototypig in Microsystems Development." *Sixth IEEE International Workshop on Rapid System Prototyping.* 10-15.

[14] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F.R. Boyer, J.P. David and G. Bois. 2002. ".Net Framework – a Solution for the next Generation Tolls for System-level Modeling and Simulation." *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI.)* 315-326.

[15] J. Liu. 2004. "A Virtual Testbed for Embedded Systems Development and Instruction." M.S. Thesis, *Dept. of Computer and Information Science Columbus: The Ohio State University.*

[16] K. Butts, W. Pree and S. Resmerita. 2012. "Chrona's validator tool suite: filling the gap between conventional software-in-the-loop and hardware-in-the-loop simulation environments." Salzburg, Austria.

[17] L. Chung and N. Subramanian. 2001. "Architecture-Based Semantic Evolution: A Study of Remotely Controlled Embedded Systems." *IEEE International Conference on Software Maintenance Proceedings,* 663-666.

[18] M.A.A. Sanvido, V. Cechticky and W. Schaufelberger. 2002. "Testing embedded control systems Using Hardware-in-the-Loop Simulation and Temporal Logic." in *15th IFAC World Congress on Automatic Control.* Barcelona, Spain.

[19] M. Barr. 1999. "Programming Embedded Systems in C and C++." Sebasopol, California:O'Reilly.

[20] M.D. Jokic and S.F. Aspkanthan. 2001. "Tethered Satellite System Models for Use in Hardware-in-the-Loop Simulation." *42nd AIAA/ASME/AsCE/AHS/ASC Structures, Sturctural Dynamics, and Materials Conference and Exhibit.* Seattle, WA.

[21] M. Gomez. 2001. "Hardware-in-the-Loop Simulation." from http://www.eetindia.co.in/STATIC/PDF/200112/EEIOL_2001DEC02_EMS_TEST_SIG_TA.pdf?SOURCES=DOWNLOAD.

[22] Microsoft, ".Net Framework" from http://msdn.microsoft.com/en-us/vstudio/aa496123.

[23] Microsoft, ".Net Framework" from http://msdn.microsoft.com/en-us/library/system.io.ports.serialport(v=vs.110).aspx.

[24] M. K. Chung and C.M. Kyung. 2004. "Improvement of compiled instruction set simulator by increasing flexibility and reducing compile time." *RSP'04: Proceeding of of the 15th IEEE International Workshop on Rapid System Prototyping.* Washington, DC, USA: IEEE Computer Society 38-44.

[25] M. Reshadi, P. Mishra and N. Dutt. 2003. "Instruction set compiled simulation: a technique for fast and flexible instruction set simulation." *Design Automation Conference, Proceedings* 758-763.

[26] M. Sanvido. 2002. "Hardware-in-the-loop simulation framework." *Automatic Control Laboratory,* ETH Zurich.

[27] P. Baracos, G. Murere, C.A. Rabbath and W. Jin. 2001. "Enabling PC-based HIL simulatoin for automotive applications." *IEEE International Electric Machines and Drives Conference* (IEMDC '01), 721-729.

[28] P. Yi, N. Abe, K. Tanaka, J. Sun and Z. Pan. 2004. "The virtual debugging system for the embedded software development." *Fourth International conference on virtual reality and its applications in industry,* vol. 5444, 357-364.

[29] S. Resmerita and W. Pree. 2012. “Verification of embedded control systems by simulation and program execution control.” *American Control Conference.* Montreal, Canada.

[30] SystemC. 2005. v2.2 Language reference manual (IEEE Std 1666-2005). *Open SystemC Initiative.* http://www.systemc.org/.

[31] Texas Instruments, TMS320F28335 Data sheet, from www.ti.com/product/tms320f28335.

[32] Texas Instruments, TMS320F28335 Serial communications interface from www.ti.com/lit/pdf/sprufz5.

[33] Texas Instruments, TMS320F28335 System control and interrupts from www.ti.com/litv/pdf/sprufb0d.

[34] Trap-gen. http://code.google.com/p/trap-gen/.

VITA

Shrikant D Labade was born in Pune in 1983 to Dagadu Labade and Gangubai Labade. He graduated from Abasaheb Garware College, Pune, in 2002 completing his intermediate education. He then joined Fergusson College, Pune to pursue Bachelors of Science (B.Sc.) in Electronics Science. He graduated from Fergusson College in 2003. He joined Master of Science at Pune University in Department Instrumentation Science in 2005. He graduated from Pune University in 2007 and joined Electronica Machine Tools Pvt. Ltd. as a "Project Engineer." He joined Center for Development of Advanced Computing (C-DAC), Hyderabad, in Feb 2009 to pursue a "Diploma in Embedded System Design." Upon completion of his diploma he joined Evincere Software Solutions in pune in January 2010. In spring 2012 he joined Department of Electrical and Computer Engineering at Louisiana State University to pursue his Master of Science in Electrical Engineering. At present he is working with Dr. Baumgartner and Dr. Ramanujam. His hobbies are hiking, reading history, autobiographies and listening to music.