2007

An architecture for embedded system communication

Chakradhar Medavarapu
Louisiana State University and Agricultural and Mechanical College

AN ARCHITECTURE FOR EMBEDDED SYSTEM COMMUNICATION

A Thesis

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

in

The Department of Computer Science

by
Chakradhar Medavarapu
B.E., in Computer Science and Engineering, University of Madras, India, 2004
May 2007

DEDICATION

To God for guiding me along the right path and for always being there for me,

my family, friends and well-wishers.

ACKNOWLEDGEMENTS

I sincerely thank my advisor Dr. Gerald Baumgartner for his constant guidance and support during the course of my thesis work. His in-depth knowledge and keen insight have helped me a lot with my thesis.

Special thanks to my friends Avinash Dundigalla and Abhishek Parakh for their constant support. I thank all other friends for their moral support.

I would like to thank Joseph Cali for his suggestions and Ms. Mildred Suares for her encouragement.

I am grateful to the Department of Computer Science for providing me with the resources to complete my thesis.

TABLE OF CONTENTS

ABSTRACT

Time is a major constraint in the development of most embedded systems. In many cases, the development of embedded software is directly dependent on the development of the embedded systems. This calls for a development framework that enables embedded software and hardware to be developed in parallel. In an attempt to solve the problem, a concept prototype hardware-in-the-loop (HIL) simulation methodology has been proposed and implemented at the Ohio State University for the TMS320LF2407A DSP board. We build on top of that HIL system by rewriting the low level device drivers that allow data and control information to be set simultaneously, thus, creating a software abstraction layer over various devices available on the DSP board. The device drivers allow data access at the processor and the pin level for the devices on the DSP board. This abstraction simulates external devices in a transparent manner using a device driver library that provides the same programming interface to the device simulators as to real devices. Also, it allows for the testing of both real and simulated hardware connected to the DSP board as a part of the embedded system. The main advantages of the framework are rapid prototyping, unit testing and monitoring. We also modify the existing serial line protocol and perform a comparison between the new and the existing protocol and show that the new protocol is efficient for large data transport. This protocol allows for the effective utilization of serial line bandwidth when the DSP board is used for signal processing or voice based applications. We present the virtual testbed as a software development tool. We conclude by exploring the future directions for the applications.

CHAPTER 1. INTRODUCTION

Embedded systems can be defined as a combination of hardware and software that form a part of some larger system and are generally designed to perform some specific task. An embedded system consists of a specially designed micro-processor, programmable ROM or RAM and other circuitry, e.g., an Application-Specific Integrated Circuits (ASICs) specific to the task for which it was designed [1, 3, 8, 9, 10 and 11].

Embedded systems have become an important part of daily life. House hold appliances, automobiles, ATMs, missile guidance systems, nuclear power plants are all controlled by embedded systems.

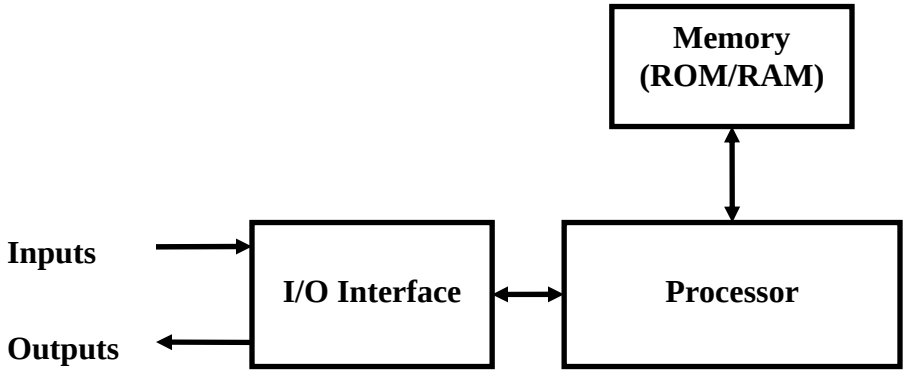

Figure 1: Block diagram of a generic embedded system (modified from [12]).

Figure 1 illustrates a generic embedded system. Excepting the general features shown in Figure 1, most embedded systems are application specific, designed to meet the requirements specific to the application. There is a vast difference in the hardware between the embedded systems used for different applications. Hence, most of the software designed for one embedded system cannot be run on another without significant modifications [12].

With the growing need for automation in various industries, the requirements for the devices that drive the automation process are increasing. Since embedded systems form a major part in driving the control systems for automation, the complexity of hardware and software in embedded systems is also increasing. Also, there is an increasing need for applications to be portable to various embedded systems with minimum effort. Porting an application code from one embedded system to another can be propelled by one or more of the following reasons:

- A hardware upgrade released by the vendor that improves the performance of the current application.
- The release of a new hardware embedded system in the market that is more suited to the current and future application requirements.
- The existing hardware has become obsolete and is no longer supported or manufactured by the vendor.

This creates a requirement for generic software frameworks. The purpose of these frameworks is to ease the transition of application code written on top the frameworks from one embedded system to another.

As any other application, embedded systems have design requirements of their own. Although, every embedded system has detailed functional requirements specific to the application, it has several requirements in common with other embedded systems. The following is an overview of the general requirements common to most embedded systems [1, 12, 13, 14]:

1. **Real time:** Embedded systems are expected to respond in real time. Code size and runtime efficiency are some of the characteristics that affect the real-time response of the embedded system.

2. **Hardware:** The hardware used in an embedded system depends on the application. The following characteristics influence the selection of hardware.
    - **Processing power:** The number of instructions executing per second measured in MIPS and the register width are some features that determine the processing power of the hardware.
    - **Memory:** The size of memory is dependant on the application and must be large enough to accommodate the executable.
    - **Power:** The energy efficiency of the hardware is critical for battery-based mobile embedded systems.
3. **Dependability:** This is an important factor that affects an embedded system design. The level of dependability required is based on the nature of the application. For example, an embedded system controlling flight avionics needs to be very dependable. Reliability, maintainability, safety and security determine the dependability of a system.
4. **Marketing:** The timeliness of the design and release of a product can be critical to get a share of the market for the application.
5. **Cost:** The following factors determine the cost of an embedded system.
    - **Expected lifetime:** It is an important characteristic that determines the cost of an embedded system. It affects all the design decisions taken during the design of an embedded system.
    - **Nature of application:** The type and nature of the application influences the cost of the system. For example, the cost of typical consumer electronics should be low as they are often produced in huge quantities and have strong competition in the market.

Given all these requirements, the embedded system needs to be designed in a cost effective manner minimizing the design time. This calls for a good design methodology that helps in achieving the above objectives. The above stated constraints can be met by designing embedded software and hardware in parallel, thus reducing the cost, system development time and the number of development cycles. Hence, an embedded system environment that allows hardware and software development in parallel is desirable [1].

Jing Liu [1] proposes and implements a virtual testbed application prototype that aims at solving the above mentioned problems. It makes use of a hardware-in-the-loop (HIL) virtual testbed to provide a realistic embedded systems development environment by utilizing an actual embedded processor (TMS320LF2407A DSP board) while simulating the external electrical and mechanical devices that are controlled by the embedded processor. Low level device drivers had two access methods where it allowed the programmer to set the data and control information separately on the hardware. The system is intended to provide a design and development environment for embedded software as realistic as possible, without physically fabricating the hardware [1].

This thesis is built on top of the work done by Jing Liu [1]. We rewrite existing low level device drivers by creating a set of functions for the TMS320LF2407A DSP board which allow data and control information to be set simultaneously, thus, creating a software abstraction layer over various devices available on the DSP board. The device drivers allow data access at the processor and the pin level for the devices on the DSP board. This abstraction allows for the testing of both real and simulated hardware, connected to the DSP board as part of the embedded system. Rapid prototyping, unit testing and monitoring are the main advantages of the abstraction framework. A detailed explanation of the advantages mentioned above can be found

in Chapter 3. This framework can be used to build higher level device abstractions that implement synchronization and threading. We also modify the existing serial line protocol making it efficient for large data transport. This allows for the effective utilization of serial line bandwidth when the DSP board is used for signal processing and voice based applications.

The rest of the thesis is organized as follows. Chapter 2 provides an overview of the related work. In Chapter 3, we discuss the HIL framework, its advantages and disadvantages. Chapter 4 presents the system setup, the implementation of the device drivers and the serial line protocol. In Chapter 5, we present the virtual testbed as a software development tool and compare the existing and the proposed embedded system design processes. We conclude in Chapter 6 and present possible future direction based on the current implementation.

CHAPTER 2. RELATED WORK

The traditional approaches employed to teach embedded systems include an embedded processor and actual electrical and mechanical devices in a laboratory environment [1, 41]. The main advantage of such a setup is that it is realistic. The main disadvantage of this setup is that it requires careful monitoring of the students as the setup involves expensive hardware. The high setup cost limits the number of students that can be educated in such an environment. Teaching dry-theory courses about DSP programming can be boring and the students may not be able to get hands-on implementation experience [1]. This calls for a balanced approach where the students get hands-on experience while lowering the laboratory setup costs. The solution to this problem is to simulate part of the systems involved in the setup while retaining the essential hardware to train students. Simulation is an efficient approach that balances the need for a low-cost yet effective training environment [1, 21]. The following paragraph supports this statement.

J Ma et al. [41] survey the existing lab setups that include real, simulated and remote labs. They draw a variety of conclusions related to the nature of learning involved and the various factors that contribute to the learning of students besides the laboratory setups. In one of their conclusions, they state that “a sense of reality can be achieved by students not only in hands-on experience, but also in virtual environments. Perhaps with the proper mix of technologies we can find solutions that meet the economic constraints of laboratories by using simulations and remote labs to reinforce conceptual understanding, while at the same time providing enough open-ended interaction to teach design” [41]. They also say that hands-on laboratories often use computers hence it is difficult to call the typical hands-on experience as real hands-on and hence the experience would actually be relative to the degrees of hands-on, simulation and remoteness being employed in the setup [41]. They also state that “the beliefs and

experiences of students may be determined more by the nature of the interfaces than by the objective reality of the laboratory technology."

The simulation is generally implemented on PCs which are common in labs today. The amount of hardware being simulated also plays an important role in determining the nature of experience of the students. While simulating external electrical and mechanical devices is essential, it has to be determined if the DSP board and other peripherals have to be simulated. Simulating the DSP board can be accomplished by specialized simulator programs generally sold by the vendors of the DSP chip or by using software such as Matlab/Simulink that specializes in rapid prototyping and in simulating the application in a pure simulation environment [1]. This leads to a complete virtual lab environment with no hands-on experience to students. Also, pure simulation is not realistic enough to understand the system completely and to evaluate the interaction between devices. All the above disadvantages call for the usage of a real embedded system with external devices being simulated.

The simulators for external devices can be built using special tools like Matlab that generate code in C or other languages based on the simulator design [15]. The generated code can be compiled with necessary modifications and can be used as a simulator or it can be used as part of a larger simulator program that uses visual components and has other functionality to analyze the output of simulation.

A simulation, in which part of the hardware connected to a system is simulated, is called Hardware-in-the-loop (HIL) simulation. HIL simulation is described in detail in Chapter 3. By employing HIL simulation, the above stated requirements like low setup cost, effective teaching and hands-on experience can be met. The usage of HIL for educational purposes is not new. W. Grega [23] uses HIL based systems for teaching a control engineering course. D. Bullock et al.

[37] use HIL for evaluating traffic control systems. They describe the usage of HIL for scientifically testing and validating traffic control algorithms. Thr drawback of this setup is its infeasibility as a teaching environment. Other applications of HIL based embedded systems are described in [37], [16], [17], [22], [27], [29], [30], [33], [35] and [38].

Modeling tools like Mathworks' Matlab Simulink are useful for high-level modeling and allow embedding code and algorithms written in various languages into components that are part of the model to test and simulate the behavior of the application [1, 15, 16]. These tools do not cover the entire embedded system design spectrum [1]. There are tools [17, 18] available for low level tasks such as code generators (which convert hardware models to VHDL and software models to C), compilers, debuggers, hardware synthesis tools at different levels (high level synthesis, logic synthesis). However, very few tools are available for system level design and testing. Some experimental and academic tools are available [19, 20], as a result of the large efforts and investments that are currently made in order to develop tools and methodologies for system level design [1].

In this thesis, we extend the work done by Jing Liu [1] on Virtual testbeds at the Ohio State University. We created a device abstraction framework for various devices available on the Texas Instruments TMS320LF2407A DSP board. A Keyhani et al. [22] proposes a virtual testbed for the design of a permanent magnet machine. It uses classical learning tools like lecture notes and computer-aided design (CAD) tools to achieve the objectives. M. Sanvido [26] proposes a framework for HIL simulation but it is implemented in the Oberon Language on the Native Oberon Operating System. Our virtual testbed, in contrast to the above approaches, uses an HIL simulator running on the Windows operating system (Windows XP). Hence the framework was written in VC# 2005. One of the main advantages of it is that it is modular in

design and the code component is reusable the details of which are described in Chapter 4. Matlab uses the JTAG interface to communicate with the DSP. We use the serial port for communication as it is easy to program unlike JTAG which does not have a well defined programming model [1]. Also, JTAG when used as the communication channel slows down the processor speed which might result in adverse affects when the system is being used as a monitoring application.

Jing Liu [1] used separate functions to distinguish between control information and data set to the devices on the DSP board. We provide a single interface to set data and control information to the devices on the DSP. J. Liu [1] does not provide an abstraction layer for all the devices on the DSP. Our abstraction layer spans over all the devices on the board. The protocol for communication between the PC and the DSP over the serial port was improved by us. The details are provided in Chapter 4. Also, our code organization is cleaner and more efficient when compared to that of [1].

CHAPTER 3. HARDWARE –IN-THE-LOOP SIMULATION

Hardware-in-the-Loop (HIL) Simulation [1, 4, 5] can be defined as a method of testing an embedded system under test (ESUT) by simulating part or all of the components of the system connected to it using a simulator.

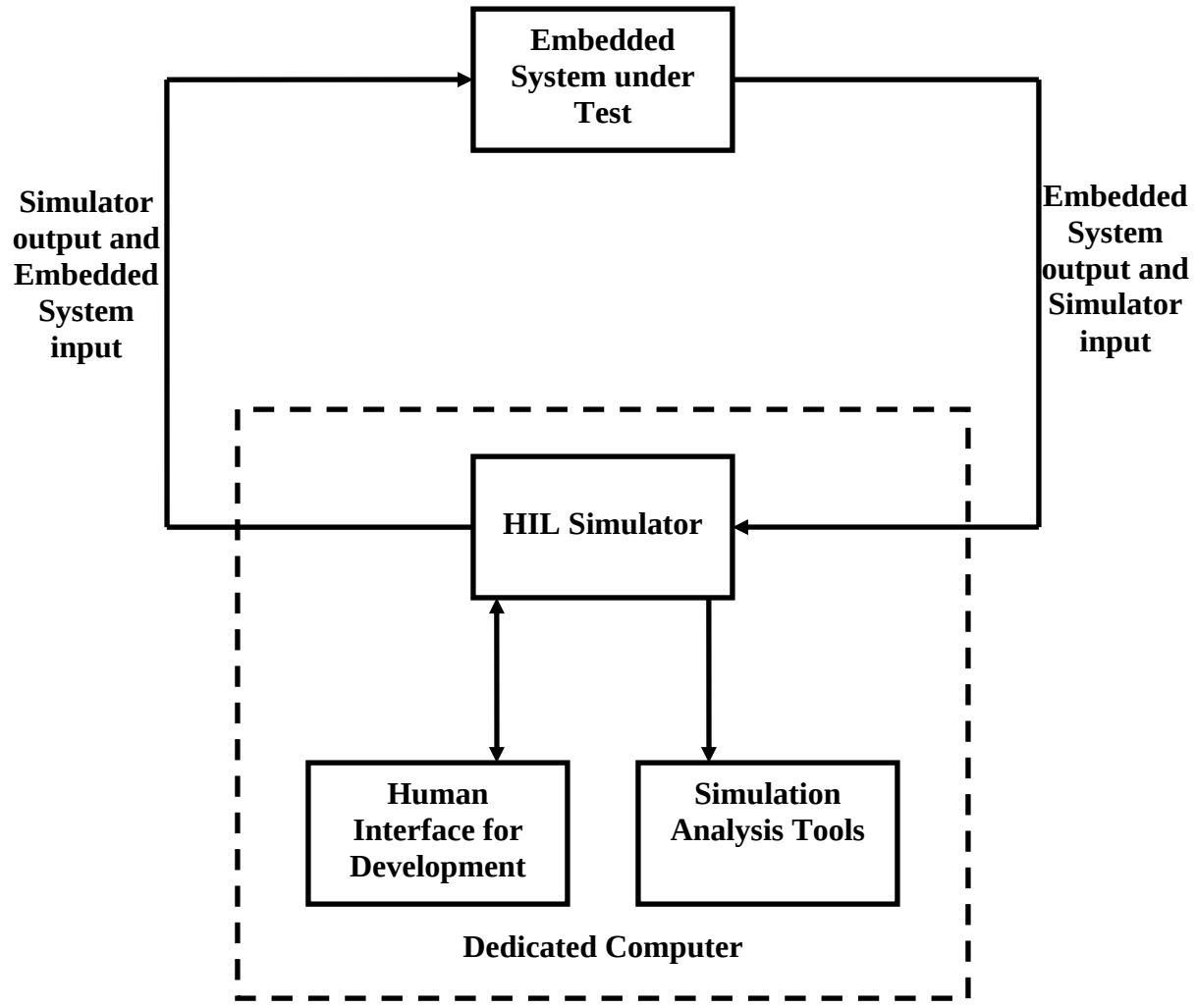

Figure 2: Block diagram of HIL simulation (modified from [1, 4, 5]).

The major difference between HIL simulation and pure simulation (where the ESUT is being simulated) is that the output from HIL simulation is in the form of electrical signals from the ESUT to the simulator while that of pure simulation are graphs and (or) other data structures. HIL simulation can be used to verify the timing aspects of the signals besides being used for functional validation [1, 26, 27].

Figure 2 shows the block diagram of Hardware-in-the-loop Simulation. The ESUT is the device being tested. The output signals from the ESUT are fed to the HIL simulator along with the simulator's control inputs. The simulator performs the simulation based on the input and outputs the signals that are fed into the ESUT along with its control inputs. Generally, the HIL simulator is a dedicated computer used by the engineer to develop and implement the simulator. The computer can also be used to host simulation analysis tools that aid in post-simulation analysis [5].

The following are the advantages of using a HIL simulation platform:

1. **Rapid prototyping:** This technique is when all the hardware connected to the ESUT as part of the system is being simulated [5].This helps in reducing the total development cycles for the ESUT and enables parallel development of various devices that are part of the system. It is particularly useful in situations where the cost of a system failure during testing is very high [1, 6]. Examples of such applications are missile, aircraft and satellite testing.
2. **Unit Testing:** In this method part of the hardware is connected to the ESUT and the remaining hardware is simulated. This technique is very helpful when some parts of the system take longer time to develop than the others. The parts developed can be attached to the ESUT while the components still in development can be simulated. For example,

when building an airplane's auto pilot system, if the control system for hydraulics is still under development while the navigation system is readily available, the navigation system can be tested with the auto pilot system simulating the hydraulic control system.

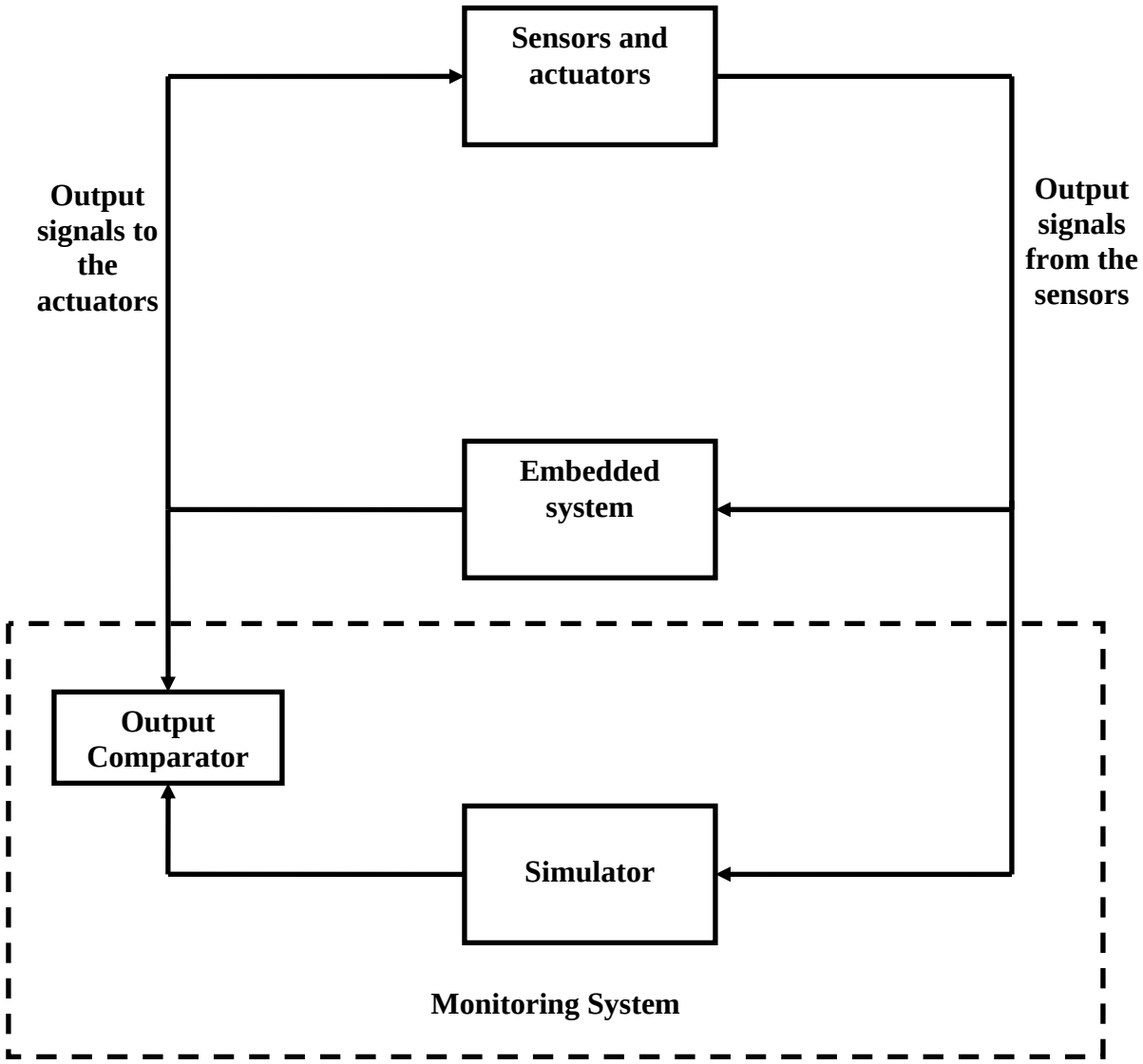

Figure 3: Block diagram of the monitoring system (modified from [1, 4]).

3. **Monitoring:** The HIL can be used for developing monitoring applications where human intervention is dangerous or is minimally required. For example, in chemical plants and

in nuclear facilities, the exposure of some chemicals and nuclear material to humans can be fatal. In order to avoid such situations, monitoring software can be used to passively monitor the system state. This is achieved by collecting the output from the feedback sensors (deployed to get system information) and the embedded controller system. The outputs from the sensors can be fed to a simulator in parallel which simulates the embedded system. The outputs from the simulator and the embedded system can then be compared to analyze the functioning of the system. Figure 3 illustrates the monitoring application.

The following are the disadvantages of using a HIL simulation platform [1]:

1. **Specialized hardware:** For simple applications, the use of regular PCs running Windows operating system might be sufficient. For complex computationally-intensive applications, server-like hardware with other specialized extensions might be required.
2. **Complex simulation software:** The software being used for analysis can be very difficult to develop. Developing such software might be very costly and time consuming as it involves developing components that simulate timing behavior between the ESUT and the simulator when performing analysis. However, the development can be considered a one-time investment as the software can be reused for future application development.
3. **Real-time requirements:** Coping with the real time requirements of the system under design can be very challenging for complex systems. This may involve the use of real time operating systems. If the modeled system is very complex, then part of its behavior may have to be approximated to get the required performance.

In most cases, the advantages of using the HIL out weigh the disadvantages as the use of HIL tends to reduce the total development time, the total cost of production.

CHAPTER 4. SYSTEM ARCHITECTURE

4.1 System Setup

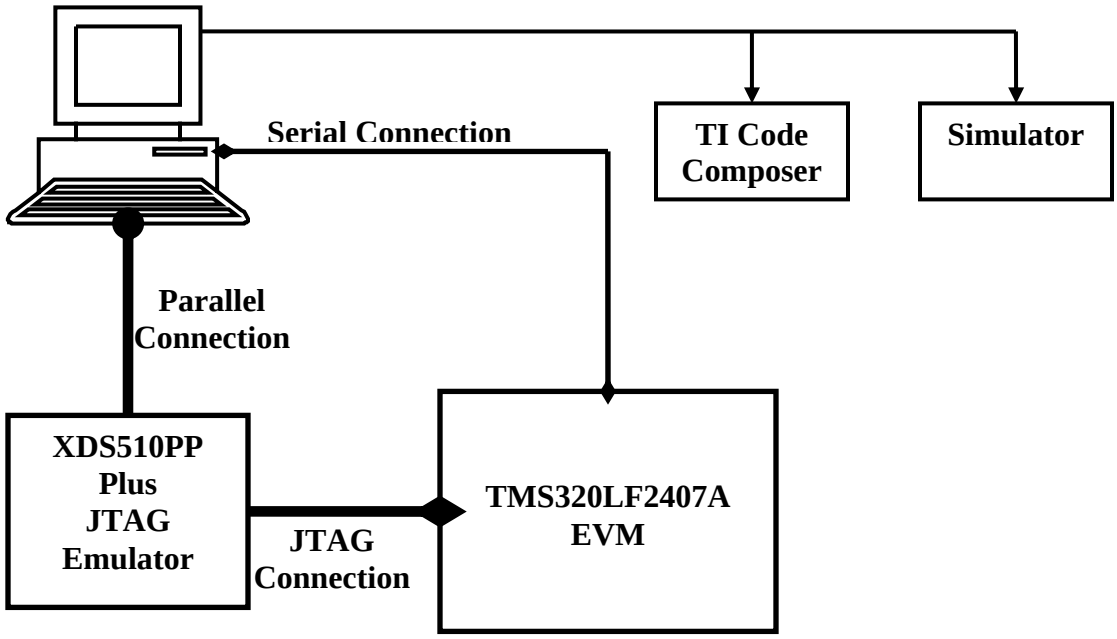

Figure 4: Block diagram of system setup (modified from [42]).

Figure 4 shows the block diagram of the system setup used for the virtual testbed project. The TMS320LF2407A DSP Evaluation Module (EVM) by Texas Instruments has been used as the embedded system. The Spectrum Digital XDS510PP Plus emulator is used to interface the PC with the EVM to load and debug the program. The emulator is connected to the PC using a parallel interface cable. The other end of the emulator is connected to the EVM through a JTAG interface. The EVM is connected to the serial port of the PC using a serial cable. This cable is used to transmit the output signals from the EVM to the PC. The PC runs the Windows XP operating system. The Texas Instruments Code Composer software is used to write, compile and download the code into the EVM. It is also used to debug the running code on the EVM. The

simulator, a GUI application, also runs on the PC. Figure 5 shows the system setup consisting of a PC running Windows XP and the EVM.

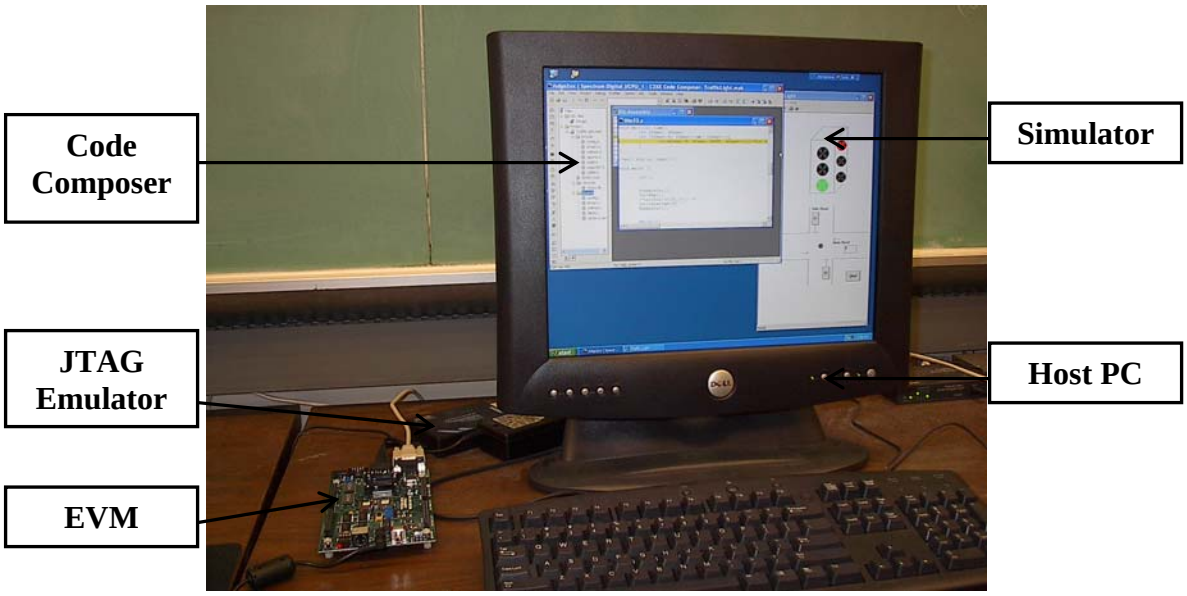

Figure 5: System Setup for Virtual Testbed

4.2 Architecture of TMS320LF2407A

The TMS320LF2407A belongs to the TMS320C2000 family of fixed point DSPs from the Texas Instruments. The TMS320LF2407A provides low-cost, low-power, and high-performance processing capabilities. Several advanced peripherals, optimized for digital motor, motion control and power conversion applications, have been integrated to provide a true single-chip DSP controller [7]. The layout of TMS320LF2407A is identical to that of its predecessor TMS320LF2407. Figure 6 shows the block diagram of TMS320LF2407.

The EVM of this processor allows full speed verification of LF2407 code, with 544 words of on-chip data memory, 32K on-chip Flash, on-chip UART, 16 channels of 10 bit on-chip Analog to Digital Conversion, 4 Channel Digital to Analog converter, Dual Event Managers, multiple PWM and capture channels on chip, four expansion connectors (data, address, I/O, and

control) to interface to any necessary evaluation circuitry, and on-board IEEE 1149.1 JTAG connection for optional Emulation [7].

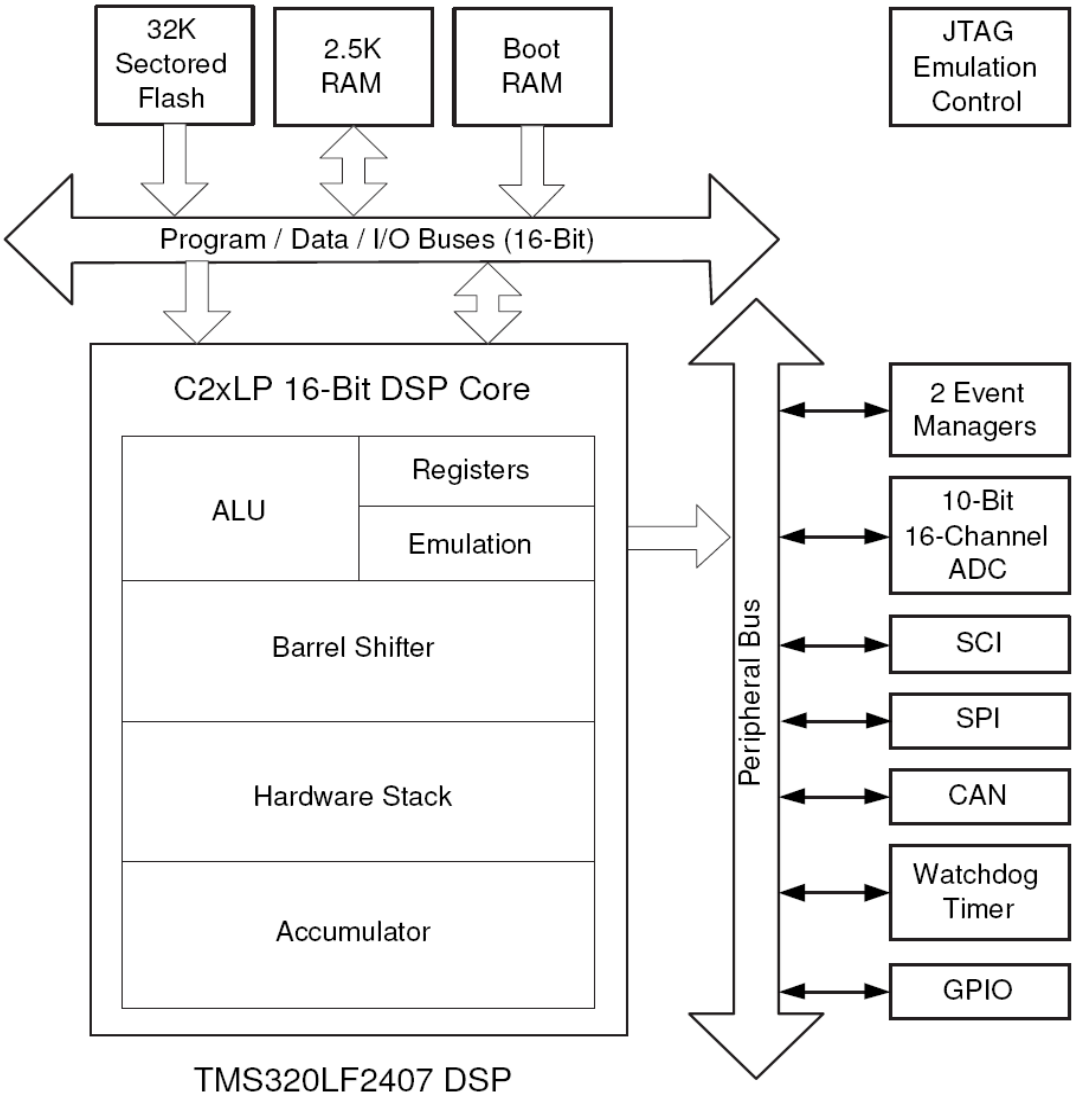

Figure 6: Block Diagram of TMS320LF2407A [1, 6].

4.3 Architecture of Virtual Testbed

Figure 7 shows the layered architecture of the Virtual Testbed. This system is analogous to the HIL simulator described in the previous chapter. The EVM is the ESUT. The user codes run on a real embedded processor on top of device drivers, either controlling real external

devices or working with the simulations of those devices running on the PC [1]. The external devices that are being simulated on the PC take inputs from the EVM over the serial line and simulate the behavior of the simulated component based on the control algorithm. The simulator can send back the computed results to the EVM. The sending back of results is directly dependent on the mode in which the simulator is being used. The modes can be rapid prototyping or monitoring.

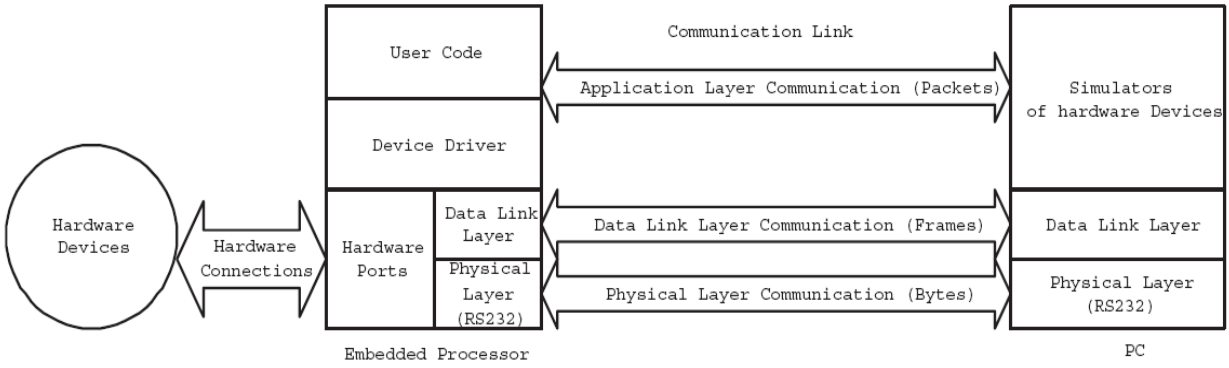

Figure 7: Layered architecture of Virtual Testbed [1].

This system is a cross platform network application between embedded processors and host machines. The embedded processors mostly do not have any operating systems running. The host machine where the code composer and simulators are running, typically run an operating system. By the principles of networking software design, we organize the system into a layered hierarchy. For our application of a localized network composed of a PC and embedded processors, we have three such layers as the Physical Link Layer, the Data Link Control Layer (DLC) and the Application Layer on each node of the network. The peer layers communicate either by a logic link or a physical link [1].

When evaluating the real-time performance requirements of the host computer that runs an HIL simulation, we would like to use a high-performance computer system. However, limited

by the available equipments for building this concept prototype, we used an ordinary PC running a non-real-time operating system — Windows XP. Such an HIL simulation system has a low cost and is relatively easy and quick to build, while running a valid and useful HIL simulation for an ESUT with low I/O rates and a simulated environment that is not overly complex. These are desirable characteristics of a system working as a concept prototype [1]. To make the system near-real time, A.Patwardhan [2] proposed and implemented near-real time serial line device drivers for the Windows XP operating system. These device drivers are adaptive in nature. We plan to include these device drivers into the system setup as our future work (refer to Chapter 6).

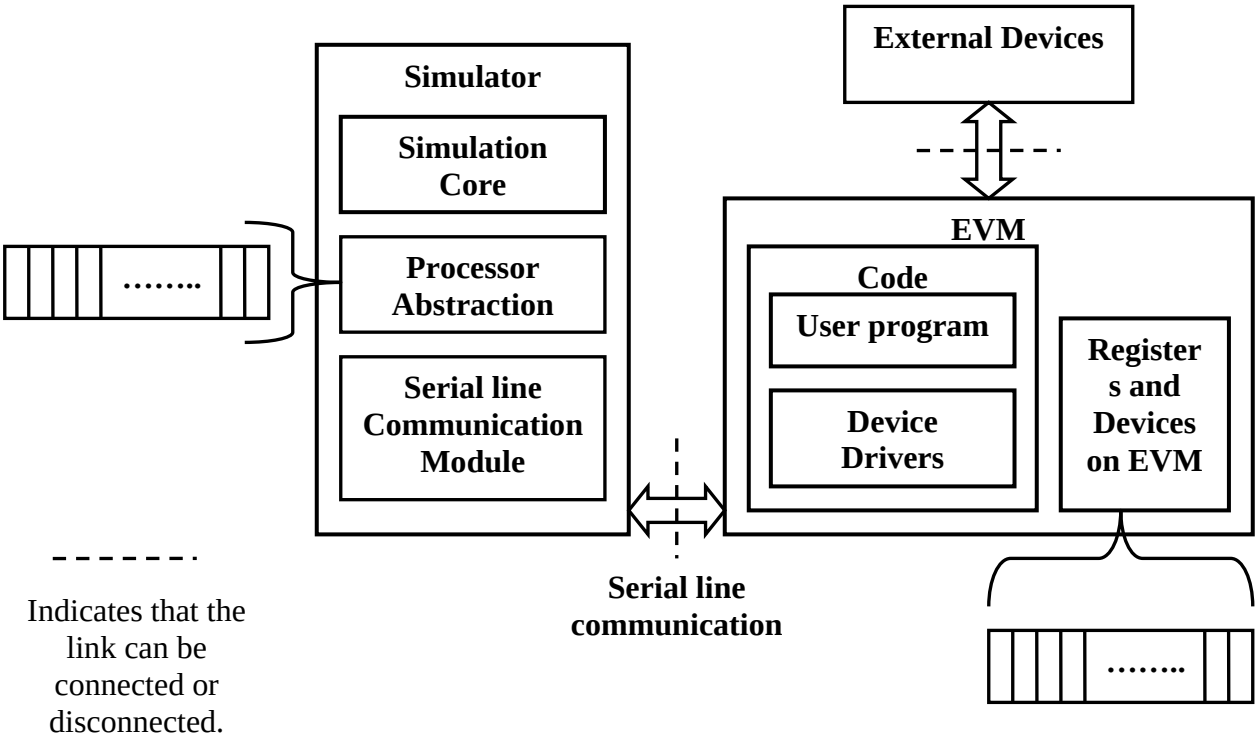

Figure 8: Detailed architecture of Virtual Testbed.

Figure 8 shows the detailed architecture of the Virtual testbed. For the sake of clarity, the underlying Windows serial port drivers are not shown in the diagram. The dotted line between

the simulator and EVM is shown for completeness of diagram. The simulator is connected to the EVM. If the connection between the EVM and Simulator is disconnected i.e., if the simulator is no longer connected to the EVM and only hardware devices are connected, such setup would not be called virtual testbed as the HIL concept would not be put to use in such a setup. If external devices are not connected to the EVM, the system resembles the rapid prototyping scenario described in chapter 3. If both, the external devices and simulator are connected to the EVM, the simulator behaves as monitoring application (refer to Chapter 3) used to monitor the outputs from the EVM.

The processor abstraction of the simulator is an abstraction of the registers and other devices on the EVM. This abstraction makes various devices and the data present in them easily accessible to the simulation core. The updation of the values to Processor abstraction from the EVM is transparent to the simulation core. The following sections describe various components present in the system and the working of the system in detail.

4.4 Software Simulators for External Devices

Simulators that run on the PC are software abstractions of hardware devices [1]. As shown in Figure 9, a simulator for the virtual testbed is composed of two major components:

- A simulation core that consist of the control algorithm for the simulator
- A Microsoft .Net component that has the following sub-components
    - Processor abstraction
    - Protocol encoder/decoder
    - The serial port communication component

The simulation core and the .Net component together can be wrapped in an optional GUI wrapper based on the type of simulator being developed. The term component here is used in a

general sense indicating a logical entity and does not necessarily represent the pre-compiled components part of .Net framework.

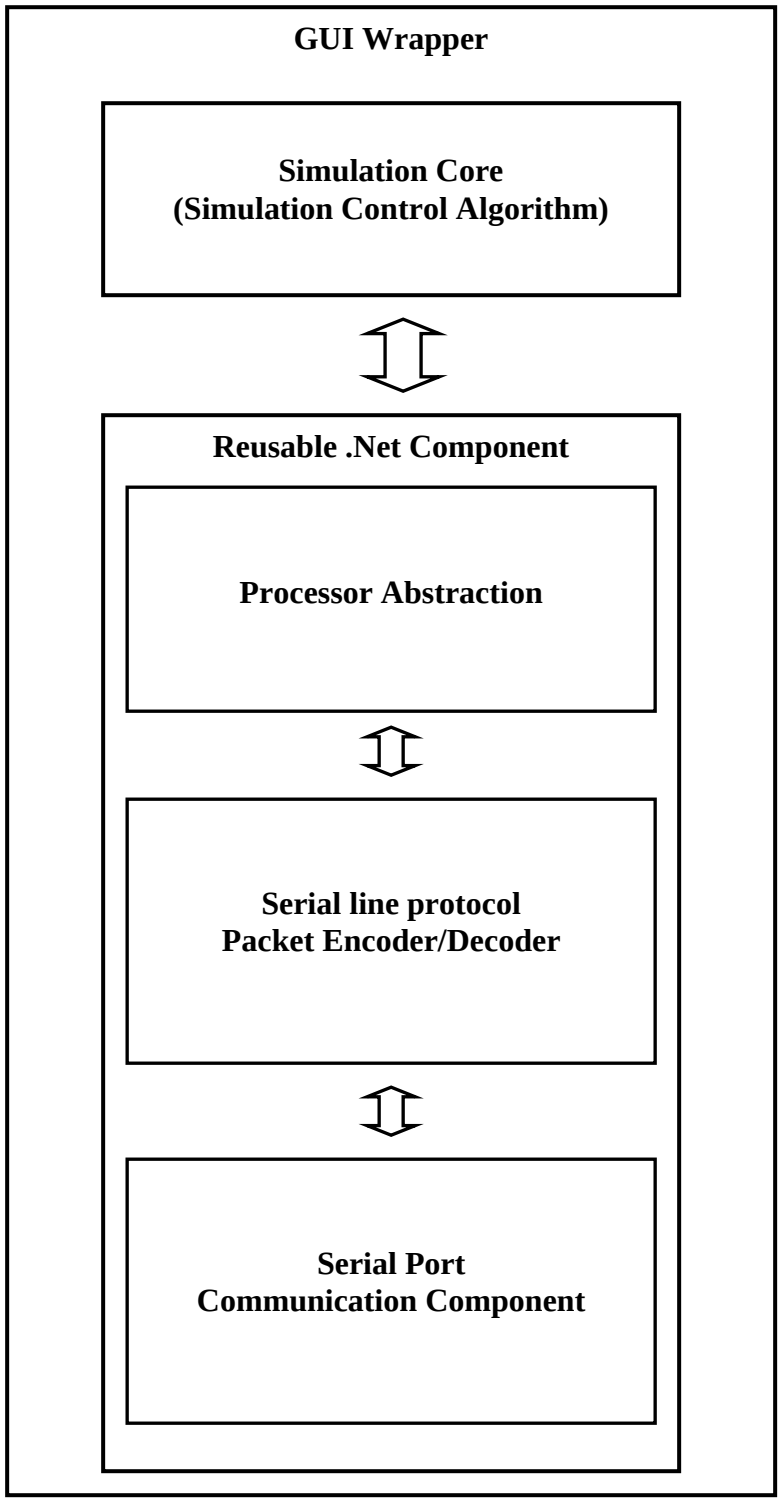

Figure 9: Simulator Components

A console based simulator does not need a GUI wrapper while a simulator with GUI components needs a GUI wrapper. The protocol encoder and decoder along with the serial port component are called as the communication module in the simulator.

When accurate timing is needed, synchronization mechanisms of the device simulation need to be introduced, such as a good estimate of the simulation frame or timestamping. The simulation frame is the length of the interval of one pass from reading in the input, computing the response, to sending the output. Figure 10 illustrates the flow of HIL simulation within one simulation frame. The simulation time proceeds with iterations of this simulation cycles, until the simulation is terminated. The simulation frame should be long enough to tolerate the worst-case time to complete all the calculations and I/O operations in the simulation. However, what we want from the device simulations is that they should mimic the behavior of the devices as realistically as possible, that is, the simulation frame time should be short enough to maintain simulation model accuracy. As the frame time is lengthened, simulation accuracy worsens. However, in addition to higher performance requirements for the simulation computer, a shorter frame time may require simplification of the simulation models so that their calculations can complete in the available time [1]. A more detailed discussion of timing events in simulation is presented in Section 4.7.

Over the years, a number of specialized programming languages and environments with library support have been developed to ease the task of developing simulation applications [1, 43]. We have used the Microsoft Visual C# for .Net 2005 to implement the simulator [44, 45]. Microsoft .Net framework is a large development platform for building next generation applications on Windows operating systems built with a primary objective of making programming easier. The .Net platform's class library is called the Framework Class Library

(FCL) which is a collection of several thousand classes most of which are wrappers for windows APIs. FCL makes programming easier by hiding the complexity of making API calls in nicely designed classes. Development through .Net saved us significant time and effort in building the GUI components for the simulator.

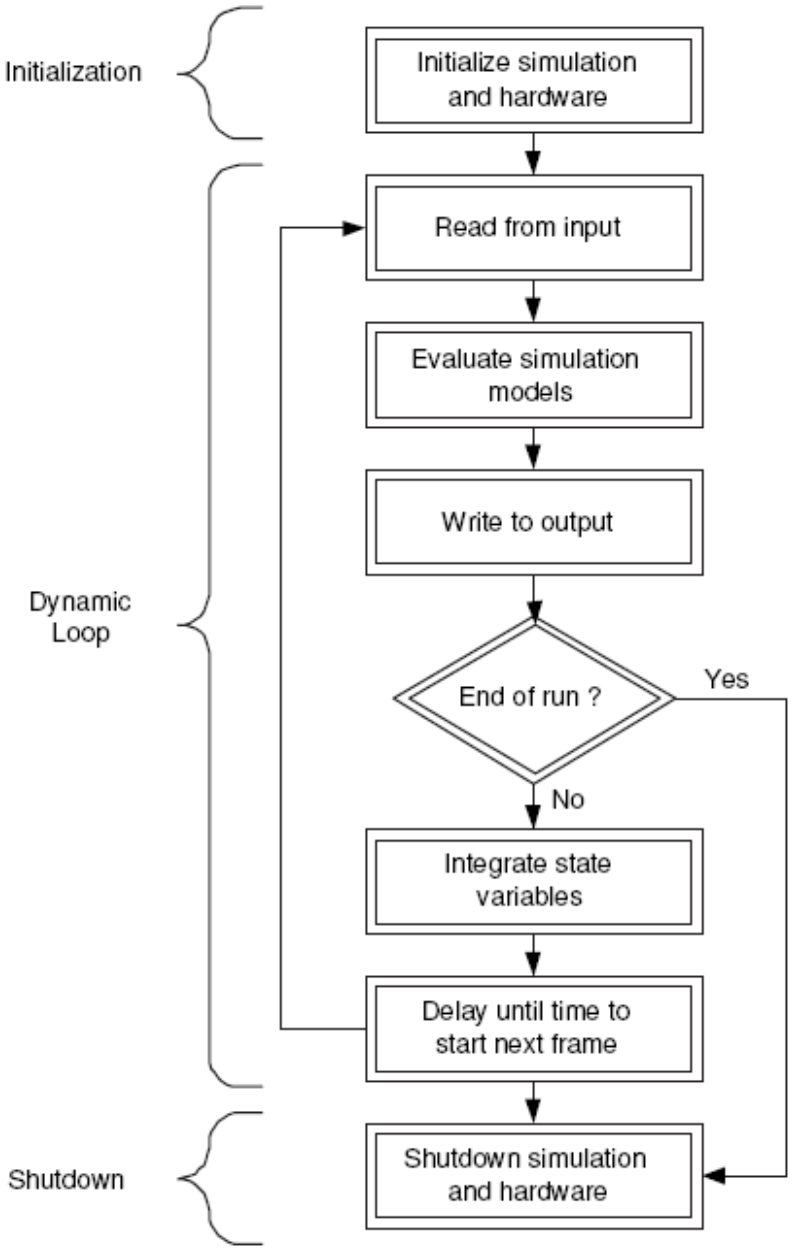

Figure 10: Program flow of a HIL Simulation Frame [1].

As the aim of the thesis is to build a concept HIL prototype for teaching purposes, the hard-real time constraints of applications developed is not of much significance. However, developing HIL simulation applications with real time constraints using general purpose programming languages like these may lead to the development of inefficient simulators.

In order to demo the HIL virtual testbed, we have implemented a traffic light controller application. Figure 12 shows the interface for the traffic light simulator application. A description of the working of the simulator is given in next section. Figure 11 gives the code for the namespace containing the processor abstraction and communication module for the simulator. The full code for methods is not listed in the figure.

J Liu [1] used Windows file abstraction for reading and writing data from/to the serial port. This was proved inefficient by [2]. Hence, we use the newly created .Net SerialPort component contained in System.IO.Ports namespace. This component provides a framework for synchronous and event-driven I/O, access to pin and break states, and access to serial driver properties [45]. These properties enabled us in building an event-driven framework for the processor abstraction. We created an event-handler that would be invoked when the data value of a register in processor abstraction changes. This is very useful when implementing the processor abstraction as part of a simulator. The simulator program can implement the handler to track the changes to the registers and execute actions accordingly.

4.5 System Behavior

The behavior of the system is explained in this section. The initial setup of the EVM board and the host PC is assumed for the description. The simulator contains code to simulate traffic, to display the traffic lights and to send messages to the EVM based on the triggered

sensors on the side streets. The traffic on the main street is simulated based on the number of vehicles per min entered into the textbox for the main road.

```
namespace DSPBoard
{   //Declaring the class for the event
    public class DSPEventArgs : EventArgs
    {   public Registers m_register;

        public DSPEventArgs(Registers register) {…}
    }

    //declaring the delegate to assist in creating the custom event
    public delegate void AbstractionUpdatedEventHandler
            (DSPBoard.DSPEventArgs e);

    class CDSPAbstraction
    {   private int[] m_TI2407ARegisters;
        //declaring the serial port
        public System.IO.Ports.SerialPort serialPort;

        //declaring the event for the delegate
        //The user who wants to know the registers that are being updated can
        // write handler for this event.
        public event AbstractionUpdatedEventHandler RegisterUpdatedEvent;

        //creates an instance of the serialPort & assigns the event handler
        private void InitializeSerialPort() {…}

        //This method is called when the event needs to be raised
        //i.e., when a register is updated
        //this method calls RegisterUpdatedEvent(new DSPEventArgs(register));
        private void DispatchRegisterUpdatedEvent(Registers register) {…}

        //event handler for data received on the serial port
        private void serialPort_DataReceived(object sender,
      System.IO.Ports.SerialDataReceivedEventArgs e) {…}

        //event handler for error received on the serial port
        private void serialPort_ErrorReceived(object sender,
      System.IO.Ports.SerialErrorReceivedEventArgs e) {…}

        //constructor to intialize Serialport and other class members
        public CDSPAbstraction() {…}

        //this returns the value of the specified register
        public int GetValue(Registers register) {…}

        //this sets the value into the specific register and sends it over
        //the serial line
        public void SetValue(Registers register, int value) {…}

        //decodes the packets and dispatches the event
        public void DecodePacket() {…}
        //Used to send the register, value pair pair over the serial line
        public void SendData(Registers register, int value) {…}
}
```

Figure 11: C# code for the .Net Component

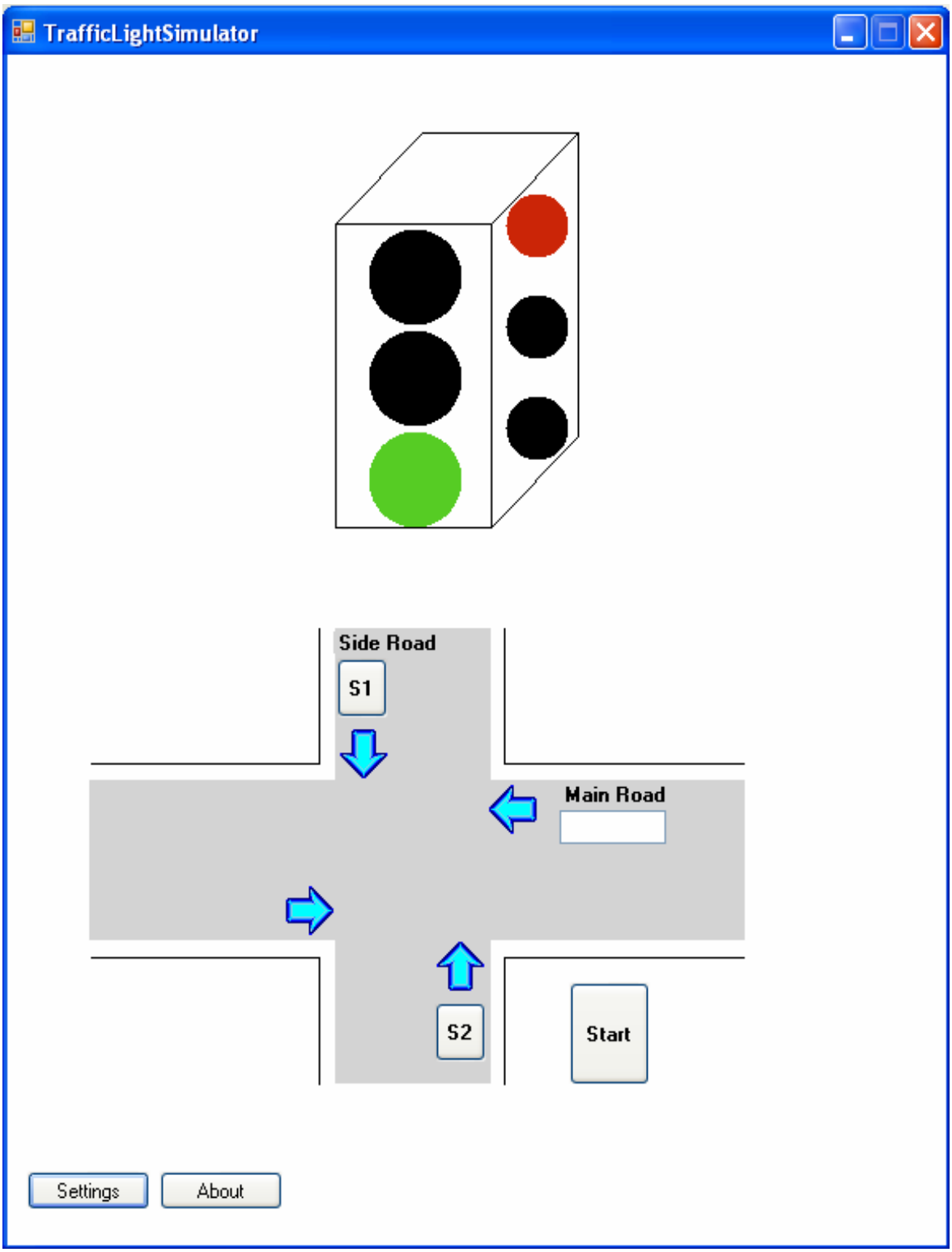

Figure 12: The interface of the Traffic light simulator.

The simulator listens over the serial line using the .Net component described earlier and based on the received messages it either updated the lights or reports error. Code Composer studio is used to write the code for the EVM in the C language. The code is complied and the program is loaded into the EVM using Code Composer. The code is transmitted to the EVM over the parallel line through the emulator. The emulator is used to load, run, halt and debug the code in the EVM.

The execution of the code inside the DSP is started from the Code composer. The EVM relays packets of data over the serial line based on the timing logic coded into the program. The protocol and drivers used for serial line communication are discussed in next sections. The simulator, which is listening over the serial line, receives the data packets, interprets the data and displays the results on the GUI. While the simulator contains packet processing and display logic, the traffic light timing logic is present on the EVM.

The traffic light stays green for the main street as long as there is no traffic on the side street. The traffic on the main street is simulated. For generating the traffic on the side street, one of the buttons may be pressed to activate the sensor. Based on the traffic of main street (which can be input through the text box before starting the simulation), the timing of green signal for the side street and the switching time from main street to side street varies.

4.6 Device Drivers

The target EVM does not have an operating system or any system level support. The primary aim of device drivers for the embedded processor is to form a platform that facilitates this application. A device driver consists of routines that provide the functionality of an I/O device. A device driver has the interface routines to perform I/O operations as well as the low

level routines that configure the I/O device and perform the actual I/O. Often it is associated with interrupt handlers to service interrupts generated by the device [1].

J Liu [1] developed device drivers for the TMS320LF2407 EVM that allowed control and data information to be set separately. Also, the drivers were not written for all the devices available on the EVM. In this thesis, we provide device drivers for all the devices available on the EVM. Our device drivers allow for setting the data and control information simultaneously to the devices on the board.

Apart from encapsulating the accesses to the I/O devices, another important aim of device drivers in this testbed is to hide from the developers the communications link with simulations. Access to devices is intercepted by device drivers. The device drivers are responsible for choosing to communicate with real devices or communicate with simulations of the devices running on the PC via a communication mechanism between the DSP and the PC. In the simulation mode, outgoing events are dispatched to the simulation by the device driver, instead of the actual I/O interface to the external devices. Incoming events from the simulation are routed through device drivers to the user codes [1].

Furthermore, the device drivers for the embedded processor can also act as an Application Programming Interface (API) platform for the developers. An API platform is a pre-defined abstraction layer above a device or system that can be used for design at a higher level. For this system, the API layer wraps the essential parts of the architecture platform: the I/O subsystem through the device drivers, and the network connection through the network communication subsystem. To maximize software reuse, the architecture platform should be abstracted at a level where the application software can use a high-level interface to the

hardware. With an API so defined, the application software can be reused for every platform instance, therefore allowing user programs to be written in a modular fashion [1].

The device drivers for the target processor were developed mainly in C, with a small amount of assembly codes where necessary. By programming in C and having the hardware details encapsulated in the device drivers of the testbed, the students are trained to apply software skills such as modular design, layered architecture, abstraction and encapsulation, etc, for developing quality embedded software [1].

One of the main aims of the device driver is to provide a consistent interface to the developer hiding the simulation details. The device driver should encapsulate the mechanism to access simulation software and real hardware. The working of the interface should be consistent irrespective of the nature of the system being used i.e. the developer should use the same function call to communicate with the simulation software and the real hardware. The inability of the user-code to distinguish between the hardware device and the simulation device is the key to minimizing the time during development phase.

The Appendix lists all the registers of the LF2407 EVM. A total of 194 registers are present in the EVM which perform various functions. We have developed the device drivers with the above stated functionality for all the registers. Figure 13 illustrates the above concept for the PADATDIR register. In Figure 13, we define the function *setPADATDIR( ).* PADATDIR is the data and direction register for I/O port A. All the registers in the EVM are memory mapped and macros are being used to assign names to the addresses.

The ‘mode’ is an extern variable which can be set by the developer based on the nature of the system. It is used to indicate the driver routine if an external device is attached to the system or if the device is being simulated. If the external hardware is being used, then the given value is

set to the PADATDIR address. If the device is being simulated, data is transmitted to the simulator based on the protocol described in next section. If both the modes are enabled, the EVM sends signals to both the external device and the simulator. In this case, the simulator can be used to monitor the functioning of the user-code.

With the new system setup, all the registers are declared as variables and they are tested by setting and getting values from the variables. By using this approach, the control logic needs to be implemented on the simulator side. It, thus, creates a processor abstraction on the simulator since one can view all the data and registers on the EVM. This approach is useful for debugging and testing purposes when all the devices are connected to the system.

A different approach can be implemented where only the required data part of a register is visible to the simulator. This reduces the control logic on the simulator. In this case, the data and control logic are implemented on the EVM. The code for this approach for the PADATDIR register is given in Figure 14. This sends out the lower byte of PADATDIR to the simulator. This approach is useful when partial set of tools or devices are connected.

The above approach to set and retrieve values of registers may not be valid to some register which perform EVM control functions. This can be explained by considering the example of SCISTL1 register which is the SCI control register 1 for the EVM. Considering the current mode as SIMULATION, if the *setSCICTL1( )* function is called to set a value, the hardware part of the set function would not be executed. Since SCICTL1 is a control register, skipping the execution of the statement *SCICTL1 = x* may lead to unexpected behavior. In order to avoid this, the *if* statement checking the mode as HARDWARE needs to be turned off. In order to accomplish this, we propose a C macro-based approach. Figure 15 shows the new format of the set function where the *if* statements are embedded in *#ifdef* macros. In this

approach, we suggest that all macros for the registers being used be defined in a configuration file as shown in Figure 16. Based on the requirements to turn the *if* statements off, the related identifiers can be undefined in the configuration file as shown in Figure 17. This approach is also extremely useful to optimize the code at compile time. Using the above technique results in the elimination of *if* statements at compile time which can compact the code considerably when a large number of registers are being used.

```
void setPADATDIR(unsigned short x)
{       //checking if hardware is attached
        if(mode&HARDWARE == 1)
                PADATDIR = x;
        //check if software simulator is being used
        if(mode&SIMULATION == 1)
        {       //follows the packet format register_id : data
                //send the data over the serial line
                   Send(PADATDIR_ID);
                   Send(x);
        }
}
```

Figure 13: A sample function to set the value of a register on the EVM

```
if(mode&SIMULATION == 1)
        {       //follows the packet format register_id: data
                //send the data over the serial line
                   Send(PADATDIR_ID);
                   Send((unsigned int)(x&0x00FF);
        }
```

Figure 14: Code showing an alternate approach to set the value of a register on the EVM

```
void setSCICTL1(unsigned short x)
{       //checking if hardware is attached
        #ifdef  SCICTL1_HWTest
        if(mode&HARDWARE == 1)
        #endif
        {
                SCICTL1 = x;
        }
        //check if software simulator is being used
        #ifdef  SCICTL1_SIMTest
        if(mode&SIMULATION == 1)
        #endif
        {       //follows the packet format register_id : data
                //send the data over the serial line
                    Send(SCICTL1_ID);
                    Send(x);
        }
}
```

Figure 15: A sample function to set the value of a register on the EVM using macros.

```
#ifdef  SCIDevices
#define SCICTL1_HWTest 1
#define SCICTL1_SIMTest 2
#define SCICTL2_HWTest 3
. . .
#endif
```

Figure 16: C code showing the declaration of a block of macros in a configuration file.

```
#undef SCICTL1_HWTest

#undef SCICTL2_HWTest
```

Figure 17: C code to undefined macros previously defined.

4.7 Timers

T A Henzinger et al. [46], propose a framework to validate embedded systems programs. They implement an interrupt driven model for ESUT where the ESUT responds to events or interrupts from external world based on timing constraints. We implement a similar model consisting of software interrupts on the ESUT that are driven by the hardware interrupts from the ESUT. We created a layer of software interrupts that handle interrupts from the hardware and from the simulator.

Three different interfaces to handle ESUT timer interrupts were provided. The following is a description of the techniques employed by the interfaces:

1. This technique addresses the scenario of invoking user code on ESUT where no communication with the simulator is required. In this method, the interrupt raised by the timer invokes a software interrupt in the code running on the DSP which calls the appropriate user-defined function.
2. This technique addresses the scenario where the simulator and the user code in the ESUT are notified. In this method, the interrupt raised by the timer sends a interrupt message to the simulator and then invokes the software interrupt pertaining to the hardware interrupt.
3. This technique addresses the scenario where the simulator takes an action and invokes the software interrupt on the ESUT. In this method, the interrupt raised by the timer is sent to the simulator. The simulator, if handling the interrupt, takes appropriate action and raises

a software interrupt on the ESUT. This mechanism of raising the software interrupt on the ESUT by the simulator is possible through the transparent communication over the serial line.

4.8 Inter-layer Communication

In [1], a message exchanged between peer application layers (simulation on PC and device drivers on the EVM) is composed of 2 to 4 consecutive fields. The following message format is used to communicate over the serial line [1]:

<Identity, Address, Value, Timestamp>.

The two fields that are included in each message are a register address field (for example, INT1, ADC_CNTL), and a field that holds the value bounded on this register. The optional identity field can be used to identify the type of the message. For example, a message could represent the information passed to some specific devices, or it could be some operator commands. The Identity field can take one of the 3 possible formats namely IO, ADC and INTERRUPT. The time stamp field is optional and can be useful when we see the need to synthetically synchronize the operations of embedded processor and simulations. This format causes delays in voice based applications consisting of stream of data packets. Also, this format does not support runtime mode changes.

To overcome these problems we modify the exiting protocol to allow for run time mode changes and to optimize the voice data transport minimizing the data transmitted over the serial line. The overview of the new protocol data packets is given in Figure 18.

To transmit data to various devices the <Id, Val> format is used where Id identifies device registers on EVM and Val represents the value set to the register. A total of 194 device registers are present on the EVM. Hence the Id can take values from 0 to 193.

1. **For transmitting data to the device registers**
    - **Format: <Id, Val>**
2. **To set streaming data for voice based applications**
    - **Format: <StreamingId, Val1, Val2,……,EscapeChar, Endmarker>**
3. **To change the mode of the device registers**
    - **Format: <Id, Val>**

Figure 18: Overview of the proposed serial line protocol.

To set streaming data for voice based applications the <StreamingId, Val1, Val2,……., EscapeChar, Endmarker> format is used where StreamingId identifies a subset of registers used for data processing. There are 24 registers part of the ADC on EVM. Hence, the StreamingId can take the values from 194 to 217. The details of the registers along with their values are given below:

- 2 ADC control registers (194, 195)
- 1 ADC max conversion register (196)
- 4 ADC channel select registers (197 to 200)
- 1 ADC auto sequence status register (201)
- 15 ADC result registers (202 to 216)
- 1 Calibration register (217)

EscapeChar is used to as an escape sequence character to distinguish between the value being transmitted and the Endmarker.

To change the mode of the device registers, the <Id, Val> format is used. The Id- identifies one of the possible 7 states for device registers on EVM. Hence Id can take the values from 218 to 224. The 7 possible states along with the values are listed below:

- HARDWARE on (218)
- SIMULATION on (219)
- HARDWARE off (220)
- SIMULATION off (221)
- HARDWARE on SIMULATION off (222)
- HARDWARE off SIMULATION on (223)
- HARDWARE on SIMULATION on (224)

The Val identifies the device registers on EVM. The Val can be any one of the 194 registers identified by numbers from 0 to 193.

Since the Id field (for format 1 and 3) and the StreamingId field (for format 2) take a total of 224 different values which is less than 255, it is possible to implement it in 1 byte which is the *unsigned char* datatype in the C language. The Val can be implemnetd as an unsigned short as most registers on the EVM are 2 bytes. In case of a register which is 1 byte in length, the lower byte of Val represents the data of the register.

4.9 Discussion

This section discusses the performance of the proposed protocol by comparing it with that of the existing protocol proposed in [1]. Performing a packet format based comparison; the existing protocol uses 5 bytes to transfer a value of 2 bytes over the serial line. The 3 bytes over head is a result of the Identity field and the Address field. The Identity field can take one of 3 possible values consuming 1 byte and the address field consists of 2 bytes. When voice based applications are considered, for every 2 bytes of voice data, 3 bytes of header information is transported across the serial line. The header information is 1.5 times the data being transmitted

which leads to wastage of serial line bandwidth. Also, the computation required to decode large volume of voice data can waste expensive processor cycles on the EVM.

The proposed protocol eliminates the overhead present in the existing protocol. To transfer a data value to a register, a total of 3 bytes of memory is required. One byte is used to indicate the Id of the register and the next 2 bytes represent the data value. The transmission of voice data is extremely efficient. It requires 1 byte to indicate the register of ADC and the number of additional bytes required to store escape sequences is directly dependent on the number of data values which are numerically equivalent to the Endmarker.

CHAPTER 5. THE VIRTUAL TESTBED AS A SOFTWARE DEVELOPMENT TOOL

In this chapter, we present the application of the virtual testbed as a software development tool. Chapter 1 describes the various constraints in the development of embedded systems. Here we discuss the methodology presented in [1] for developing hardware and software in parallel using HIL techniques.

Figure 19, summarizes the current design methodology in embedded system development. Starting from some informal specification of functionality and a set of constraints (time and power constraints, cost limits, etc.), a formal specification of the system is generated, including the choice of a microprocessor and peripheral devices. From the specifications from stage 1, the hardware is built. At this stage, the development of embedded software is begun. The design of embedded software involves considerable amount of time. The various stages involved in developing the software are shown in Figure 20. After the software is developed, it is highly probable that the first releases of the software embedded in them will fail to meet the constraints and might even damage the hardware. In this case, the engineers will have to go back and modify the specification of the system, for instance, to modify the existing architecture, and start another round of this process [1].

The use of HIL simulation eliminates expensive and lengthy iterations in machining and fabrication of parts, and speeds development towards a more efficient design. The HIL systems allows for parallel development of hardware and software. Figure 21 shows the various steps involved in embedded system development through virtual testbed. The various advantages of using this approach is explained below [1]:

- The development of software and hardware is done in parallel. This provides flexibility in developing software and eliminates the “destructive testing” of the hardware developed.
- The development of software would be independent of the development of hardware. This eliminates any dependency on software development that is a result of the hardware development loop in Figure 19. This is very useful for complex systems which are expensive and are difficult to build and test, as the hardware development loop requires considerable amount of time.

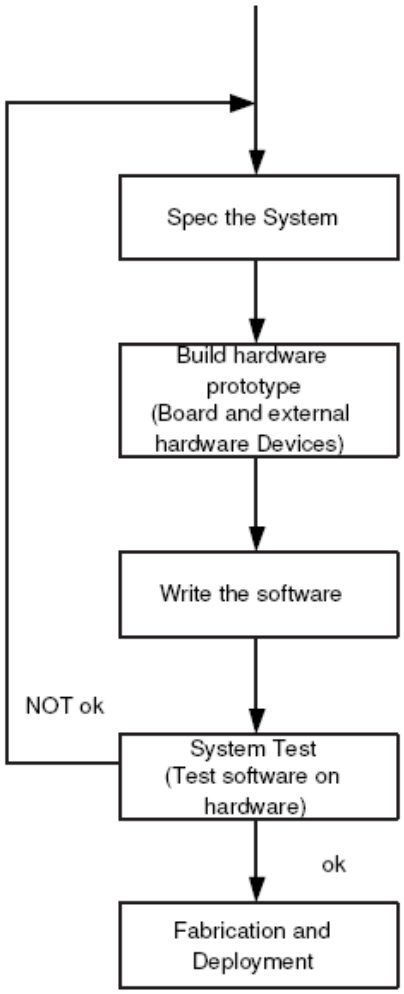

Figure 19: Current Embedded System Development Design Flow [1].

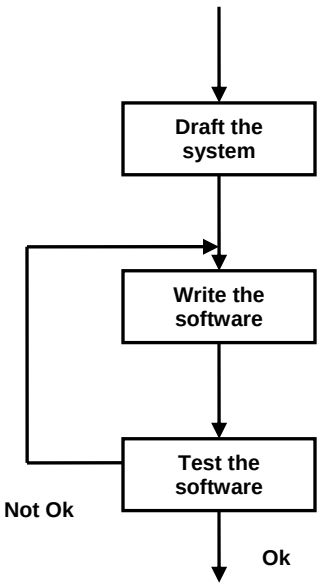

Figure 20: The stages in writing the software.

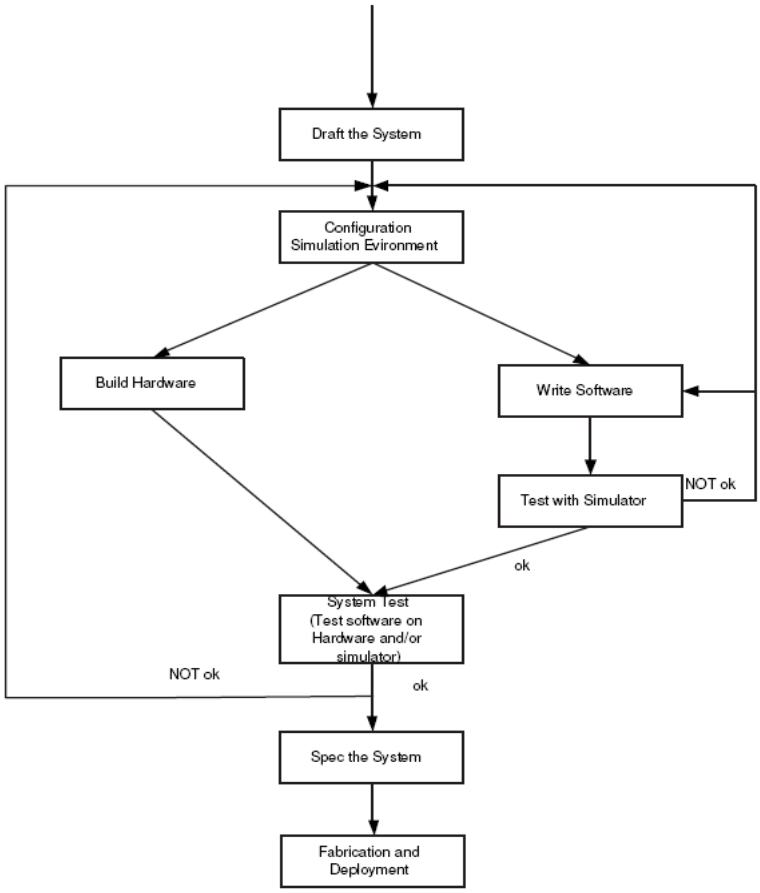

Figure 21: Proposed Embedded System Development Design Flow [1].

CHAPTER 6. CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this thesis, we have presented a framework that simulates external devices in a transparent manner using a device driver library that provides the same programming interface to the device simulators as to real devices. The device drivers have been designed to allow data access at the processor and the pin level for the devices on the DSP board. Also, they allows for the testing of both real and simulated hardware, connected to the DSP board as a part of the embedded system. We have described the effectiveness of this framework for rapid prototyping, unit testing and monitoring.

We have modified the existing serial line protocol and performed a comparison between the new and the existing protocol and prove that the new protocol is efficient for large data transport. The new protocol allows for the effective utilization of serial line bandwidth when the EVM is used for signal processing and voice based applications.

6.2 Future Work

The current implementation of the HIL concept prototype is non real-time as it runs on a non real-time operating system on an ordinary PC. Also, the usage of general purpose programming languages for designing the simulator limits the effectiveness of the software developed for special purpose simulations as these tools, APIs are not designed for time constrained systems. However, the setup can still be used to run a valid simulation for an ESUT with low I/O rates and a simulated environment that is not overly complex. To run high performance real-time simulations, the setup requires a high performance CPU with support for high I/O rates and with support for real time operations [1]. Listed below is a summary of various techniques that can be used to improve the performance of the current system:

- **Use of a Real-Time Operating System:** A real time operating system (RTOS) or a real-time software framework can be used to achieve the real-time characteristics for the simulations. An RTOS is an operating system where interrupts are guaranteed to be handled within a certain specified maximum time, thereby making it suitable for control of hardware in embedded systems and other time-critical applications. Therefore, with an RTOS, the requests from our application testbed can be guaranteed to be accomplished within a certain pre defined time limit [1].
- **Use of Serial line drivers that support real time communication:** A.Patwardhan [2] proposed and implemented near real-time serial line device drivers for the Windows XP operating system. The device drivers are adaptive in nature. These device drivers can be included into the existing system setup to provide real-time support.
- **Runtime configuration support for the simulator:** Runtime configuration support to configure and connect pins of the processor on DSP with special software component on the simulator using a GUI can be provided. This provides fine grained control to simulators as data from individual pins can be diverted at runtime to appropriate software component.
- **Web based remote simulation:** The current application is a standalone application on a PC for a single user. This can be extended to multiple users by running the simulator code on a web server as a web service or as a DLL which is connected to a server side component to listen from various clients and simulate the result based on the input and return the results of the simulation as a graphical web page.
- **Automated testing of the event sequence with timing constraints:** A testing tool to perform automated testing of the event sequence with timing constraints can be created. The

aim of such a tool is to write a test harness to test the behavior of the application under timing constraints.

REFERENCES

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

[2] A. Patwardhan, "An Architecture for Adaptive Real Time Communication with Embedded Devices," in *Dept. of Computer Science* Baton Rouge: Louisiana State University, 2006.

[3] Wikipedia, "Embedded System," from http://en.wikipedia.org/wiki/Embedded_system.

[4] N. Instruments, "LabVIEW FPGA in Hardware-in-the-Loop Simulation Applications," from http://zone.ni.com/devzone/cda/tut/p/id/3567.

[5] D. S. Cocanougher, S. A. Mirsky, and D. B. Card, "Application of Hardware-in-the-Loop Simulation to operational test and Evaluation," *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing IV,* vol. 3697, pp. 80-88, 1999.

[6] F. Bormann, "16-Bit-DSP-Microcontroller Texas Instruments TMS320LF2407."

[7] T. Instruments, TMS320LF2407A Data Sheet, from http://focus.ti.com/lit/ds/symlink/tms320lf2407a.pdf.

[8] Q. Li and C. Yao, *Real-Time Concepts for Embedded Systems*. California: CMP Books, 2003.

[9] T. Noergaard, *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Oxford, England: Newnes, 2005.

[10] R. S. Janka, *Specification and Design Methodology for Real-Time Embedded Systems*. Netherlands: Kluwer Academic Publishers, 2002.

[11] G. R. Wilson, *Embedded Systems and Computer Architecture*. Oxford, England: Newnes, 2001.

[12] M. Barr, *Programming Embedded Systems in C and C++*. Sebastopol, California: O'Reilly, 1999.

[13] P. Marwedel, *Embedded System Design*. Netherlands: Springer, 2005.

[14] S. Gruden, "Efficient Development of high Quality Software for Embedded Systems," *Informacije MIDEM,* vol. 33, pp. 260-266, 2003.

[15] Mathworks, "Simulink," from www.mathworks.com/products/simulink/.

[16] A. A. Reyes, A. P. Narayanasamy, and A. Dogan, "Simulation-based Development of Real-time, Embedded Software for cooperative, autonomous Aerial Vehicles," in *22nd Digital Avionics Systems Conference. Proceedings*. vol. 2, 2003, pp. 8.A.3-1-11

[17] P. Yi, N. Abe, K. Tanaka, J. Sun, and Z. Pan, "The Virtual Debugging System for Embedded Software Development," in *Fourth International conference on virtual reality and its applications in industry* vol. 5444, 2004, pp. 357-364.

[18] M. L. Rebaiaia, M. Benmohamed, J. M. Jaam, and A. Hasnah, "A Toolset for the Specification and Verification of Embedded Systems," in *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2003)*. vol. 4 Computer Science Dept., University of Batna, Algeria, 2003, pp. 1539–1545

[19] F. Wang, "Formal Verification of Timed Systems: A Survey and Perspective," *Proceedings of the IEEE,* vol. 92, pp. 1283-1305, August 2004.

[20] G. Berry, "Synchronous Methodology for Designing Hardware, Software and Mixed Embedded Systems," in *Proceedings of 17th International Conference on VLSI Design*, 2004, pp. 24-25.

[21] B. L. Ballard, R. E. Elwell, R. C. Gettier, F. P. Horan, A. F. Krummenoehl, and D. B. Schepleng, "Simulation Approaches for supporting Tactical System Development," *Johns Hopkins APL technical digest* vol. 23, pp. 311-324, 2002.

[22] A. Keyhani and A. B. Proca, "A Virtual Testbed for lnstruction and Design of Permanent Magnet Machines," *IEEE Transactions on Power Systems*, vol. 14, pp. 795 - 801 1999.

[23] W. Grega, "Hardware-in-the-Loop Simulation and its Application in Control Education," in *29th ASEE/IEEE Frontiers in Education Conference*, 1999, pp. 12b6-7-12.

[24] E. Carpanzano, L. Ferrarini, C. Maffezzoni, A. Cataldo, and G. Ceiner, "Testing Industrial Distributed Control Systems with Hardware-in-the-Loop Simulators," in *Simulation in Industry'99. 11th European Simulation Symposium 1999. ESS'99* Erlangen-Nuremberg, Germany 1999, pp. 574-578.

[25] DSpace, from www.dspaceinc.com.

[26] M. Sanvido, "Hardware-in-the-Loop Simulation Framework," Automatic Control Laboratory, ETH Zurich.

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

[28] MathTools, "Control and Systems Modeling," from www.mathtools.net.

[29] M. Gomez, "Hardware-in-the-Loop Simulation," from http://www.embedded.com/story/OEG20011129S0054.

[30] "Hardware in the Loop Traffic Simulation," Transport Research Center, University of Florida. March 2005.

[31] M. A. Wehrmeister, C. E. Pereira, and L. B. Becker, "Optimizing the Generation of Object-Oriented Real-Time Embedded Applications Based on the Real-Time Specification for Java," in *Design, Automation and Test in Europe (DATE '06)*, 2006, pp. 1-6.

[32] K. Ramamritham, K. Arya, and G. Fohler, "System Software for Embedded Applications," in *Proceedings of 17th International Conference on VLSI Design*, 2004, pp. 12-14.

[33] H.-J. Herpel, M. Glesner, H. Eggert, W. Suss, M. Gorges-Schleuter, and W. Jakob, "Rapid Prototyping in Microsystems Development," in *Sixth IEEE International Workshop on Rapid System Prototyping*, 1995, pp. 48 - 53

[34] M. Grünewald, Jörg-Christian, and N. U. Rückert, "A Performance Evaluation Method for optimizing Embedded Applications," in *The 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, 2003, pp. 10-15.

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

[36] M. D. Jokic and S. F. Asokanthan, "Tethered Satellite System Models for Use in Hardware-in-the-Loop Simulations," in *42nd AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics, and Materials Conference and Exhibit* Seattle, WA, 2001.

[37] D. Bullock and T. Urbanik, "Hardware-In-The-Loop Evaluation Of Traffic Signal Systems," in *Tenth International Conference on Road Transport Information and Control*, 2000, pp. 177 - 181

[38] P. Baracos, G. Murere, C. A. Rabbath, and W. Jin, "Enabling PC-Based HIL Simulation for Automotive Applications," in *IEEE International Electric Machines and Drives Conference (IEMDC '01)*, 2001, pp. 721-729.

[39] C. Angelov and K. Sierszecki, "A Software Framework for Component-based Embedded Applications," in *11th Asia-Pacific Software Engineering Conference* 2004, pp. 655 - 662

[40] S. Mansoor, "Behaviour and Operation of Pump Storage Hydro Plants," in *School of Informatics* Bangor: University of Wales, 2000.

[41] J. Ma and J. V. Nickerson, "Hands-on, Simulated, and Remote laboratories: A Comparative Literature Review," *ACM Computing Surveys (CSUR)* vol. 38, September 2006.

[42] T. I. Korea, "USB JTAG Emulator," from http://www.tikorea.co.kr/usb.html.
.

[43] M. A. WIlson, "Visualization in Hardware-in-the-Loop Simulation," *Technologies for Synthetic Environments: Hardware-in-the-Loop Testing II*, vol. 3084, pp. 358-363, July 1997.

[44] J. Lapalme, E. M. Aboulhamid, G. Nicolescu, L. Charest, F. R. Boyer, J. P. David, and G. Bois, ".Net Framework - a Solution for the next Generation Tools for System-level Modeling and Simulation," in *Proceedings. Design, Automation and Test in Europe Conference and Exhibition*, 2004, pp. 732-733.

[45] Microsoft, ".Net Framework," from http://msdn2.microsoft.com/en-us/netframework/default.aspx.

[46] T. A. Henzinger and C. M. Kirsch, "The Embedded Machine: Predictable, Portable Real-Time Code," in *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, 2002, pp. 315-326.

APPENDIX: REGISTER LIST

The following is a listing of all the registers with their addresses available on the LF2407A EVM as a C language header file:

```
#define IMR             *(volatile unsigned int *)0x0004    /* CPU Interrupt
Mask Register */
#define GREG            *(volatile unsigned int *)0x0005    /* Global Data
Memory Configuration Register */
#define IFR             *(volatile unsigned int *)0x0006    /* CPU Interrupt
Flag Register */
#define PIRQR0          *(volatile unsigned int *)0x7010    /* Peripheral
Interrupt Request Register 0 */
#define PIRQR1          *(volatile unsigned int *)0x7011    /* Peripheral
Interrupt Request Register 1 */
#define PIRQR2          *(volatile unsigned int *)0x7012    /* Peripheral
Interrupt Request Register 2 */
#define PIACKR0         *(volatile unsigned int *)0x7014    /* Peripheral
Interrupt Acknowledge Register 0 */
#define PIACKR1         *(volatile unsigned int *)0x7015    /* Peripheral
Interrupt Acknowledge Register 1 */
#define PIACKR2         *(volatile unsigned int *)0x7016    /* Peripheral
Interrupt Acknowledge Register 2 */
#define SCSR1           *(volatile unsigned int *)0x7018    /* System Control
and Status Register 1   */
#define SCSR2           *(volatile unsigned int *)0x7019    /* System Control
and Status Register 2   */
#define DINR            *(volatile unsigned int *)0x701C    /* Device
Identification Name Register  */
#define PIVR            *(volatile unsigned int *)0x701E    /* Peripheral
Interrupt Vector Register */
#define WDCNTR          *(volatile unsigned int *)0x7023    /* Watchdog
Counter Register              */
#define WDKEY           *(volatile unsigned int *)0x7025    /* Watchdog Reset
Key Register                  */
#define WDCR            *(volatile unsigned int *)0x7029    /* Watchdog Timer
Control Register        */
#define SPICCR          *(volatile unsigned int *)0x7040    /* SPI
Configuration Control Register      */
#define SPICTL          *(volatile unsigned int *)0x7041    /* SPI Operation
Control Register        */
#define SPISTS          *(volatile unsigned int *)0x7042    /* SPI Status
Register                            */
#define SPIBRR          *(volatile unsigned int *)0x7044    /* SPI Baud Rate
Control Register        */
#define SPIRXEMU  *(volatile unsigned int *)0x7046    /* SPI Emulation Buffer
Register          */
#define SPIRXBUF  *(volatile unsigned int *)0x7047    /* SPI Serial Receive
Buffer Register   */
#define SPITXBUF  *(volatile unsigned int *)0x7048    /* SPI Serial Transmit
Buffer Register   */
```

```
#define SPIDAT          *(volatile unsigned int *)0x7049    /* SPI Serial
Data Register                       */
#define SPIPRI          *(volatile unsigned int *)0x704F    /* SPI Priority
Control Register        */
#define SCICCR          *(volatile unsigned int *)0x7050    /* SCI
Communication Control Register      */
#define SCICTL1         *(volatile unsigned int *)0x7051    /* SCI Control
Register 1                  */
#define SCIHBAUD  *(volatile unsigned int *)0x7052    /* SCI Baud-Select
Register, High-Bits     */
#define SCILBAUD  *(volatile unsigned int *)0x7053    /* SCI Baud-Select
Register, Low-Bits      */
#define SCICTL2         *(volatile unsigned int *)0x7054    /* SCI Control
Register 2                  */
#define SCIRXST         *(volatile unsigned int *)0x7055    /* SCI Receiver
Status Register             */
#define SCIRXEMU  *(volatile unsigned int *)0x7056    /* SCI Emulation Data
Buffer Register   */
#define SCIRXBUF  *(volatile unsigned int *)0x7057    /* SCI Receiver Data
Buffer Register   */
#define SCITXBUF  *(volatile unsigned int *)0x7059    /* SCI Transmit Data
Buffer Register   */
#define SCIPRI          *(volatile unsigned int *)0x705F    /* SCI Priority
Control Register        */
#define XINT1CR         *(volatile unsigned int *)0x7070    /* External
Interrupt 1 Control Register*/
#define XINT2CR         *(volatile unsigned int *)0x7071    /* External
Interrupt 2 Control Register*/
#define MCRA            *(volatile unsigned int *)0x7090    /* I/O Mux
Control Register A                  */
#define     MCRB        *(volatile unsigned int *)0x7092    /* I/O Mux
Control Register B                  */
#define MCRC            *(volatile unsigned int *)0x7094    /* I/O Mux
Control Register C                  */
#define PEDATDIR  *(volatile unsigned int *)0x7095    /* I/O PORT E Data and
Direction Register      */
#define PFDATDIR  *(volatile unsigned int *)0x7096    /* I/O PORT F Data and
Direction Register      */
#define PADATDIR  *(volatile unsigned int *)0x7098    /* I/O Port A Data and
Direction Register      */
#define PBDATDIR  *(volatile unsigned int *)0x709a    /* I/O Port B Data and
Direction Register      */
#define PCDATDIR  *(volatile unsigned int *)0x709c    /* I/O Port C Data and
Direction Register      */
#define PDDATDIR  *(volatile unsigned int *)0x709e    /* I/O Port D Data and
Direction Register      */
#define ADCCTRL1  *(volatile unsigned int *)0x70a0    /* ADC Control Register
1                           */
#define ADCCTRL2  *(volatile unsigned int *)0x70a1    /* ADC Control Register
2                           */
#define MAXCONV         *(volatile unsigned int *)0x70a2    /* ADC Maximum
Conversion Register             */
#define CHSELSEQ1 *(volatile unsigned int *)0x70a3    /* ADC Channel Select
Sequencing Control 1    */
#define CHSELSEQ2 *(volatile unsigned int *)0x70a4    /* ADC Channel Select
Sequencing Control 2    */
```

```
#define CHSELSEQ3 *(volatile unsigned int *)0x70a5    /* ADC Channel Select
Sequencing Control 3    */
#define CHSELSEQ4 *(volatile unsigned int *)0x70a6    /* ADC Channel Select
Sequencing Control 4    */
#define AUTO_SEQ_SR     *(volatile unsigned int *)0x70a7    /* ADC Auto
Sequence Status Register            */
#define RESULT0         *(volatile unsigned int *)0x70a8    /* ADC Result
Register 0                          */
#define RESULT1         *(volatile unsigned int *)0x70a9    /* ADC Result
Register 1                          */
#define RESULT2         *(volatile unsigned int *)0x70aa    /* ADC Result
Register 2                          */
#define RESULT3         *(volatile unsigned int *)0x70ab    /* ADC Result
Register 3                          */
#define RESULT4         *(volatile unsigned int *)0x70ac    /* ADC Result
Register 4                          */
#define RESULT5         *(volatile unsigned int *)0x70ad    /* ADC Result
Register 5                          */
#define RESULT6         *(volatile unsigned int *)0x70ae    /* ADC Result
Register 6                          */
#define RESULT7         *(volatile unsigned int *)0x70af    /* ADC Result
Register 7                          */
#define RESULT8         *(volatile unsigned int *)0x70b0    /* ADC Result
Register 8                          */
#define RESULT9         *(volatile unsigned int *)0x70b1    /* ADC Result
Register 9                          */
#define RESULT10  *(volatile unsigned int *)0x70b2    /* ADC Result Register
10                          */
#define RESULT11  *(volatile unsigned int *)0x70b3    /* ADC Result Register
11                          */
#define RESULT12  *(volatile unsigned int *)0x70b4    /* ADC Result Register
12                          */
#define RESULT13  *(volatile unsigned int *)0x70b5    /* ADC Result Register
13                          */
#define RESULT14  *(volatile unsigned int *)0x70b6    /* ADC Result Register
14                          */
#define RESULT15  *(volatile unsigned int *)0x70b7    /* ADC Result Register
15                          */
#define CALIBRATION     *(volatile unsigned int *)0x70b8    /* ADC
Calibration Register                        */
#define CAN_MDER  *(volatile unsigned int *)0x7100    /* CAN Mailbox
Direction/Enable Register  */
#define CAN_TCR         *(volatile unsigned int *)0x7101    /* CAN
Transmission Control Register */
#define CAN_RCR         *(volatile unsigned int *)0x7102    /* CAN Receiver
Control Register        */
#define CAN_MCR         *(volatile unsigned int *)0x7103    /* CAN Master
Control Register            */
#define CAN_BCR2  *(volatile unsigned int *)0x7104    /* CAN Bit
Configuration Register 2            */
#define CAN_BCR1  *(volatile unsigned int *)0x7105    /* CAN Bit
Configuration Register 1            */
#define CAN_ESR         *(volatile unsigned int *)0x7106    /* CAN Error
Status Register             */
#define CAN_GSR         *(volatile unsigned int *)0x7107    /* CAN Global
Status Register             */
```

```
#define CAN_CEC         *(volatile unsigned int *)0x7108    /* CAN Error
Counter Register              */
#define CAN_IFR         *(volatile unsigned int *)0x7109    /* CAN Interrupt
Flag Register                 */
#define CAN_IMR         *(volatile unsigned int *)0x710a    /* CAN Interrupt
Mask Register                 */
#define CAN_LAM0_H      *(volatile unsigned int *)0x710b    /* CAN Local
Acceptance Mask MB0 and MB1 high */
#define CAN_LAM0_L      *(volatile unsigned int *)0x710c    /* CAN Local
Acceptance Mask MB0 and MB1 low  */
#define CAN_LAM1_H      *(volatile unsigned int *)0x710d    /* CAN Local
Acceptance Mask MB3 and MB2 high */
#define CAN_LAM1_L      *(volatile unsigned int *)0x710e    /* CAN Local
Acceptance Mask MB3 and MB2 low  */
#define CAN_MSGID0L     *(volatile unsigned int *)0x7200    /* CAN Message ID
for MB0 , low           */
#define CAN_MSGID0H     *(volatile unsigned int *)0x7201    /* CAN Message ID
for MB0 , high    */
#define CAN_MSGCTRL0 *(volatile unsigned int *)0x7202 /* CAN Message Control
Field 0           */
#define CAN_MBX0A *(volatile unsigned int *)0x7204    /* CAN 2 of 8 Bytes of
Mailbox 0   */
#define CAN_MBX0B *(volatile unsigned int *)0x7205    /* CAN 2 of 8 Bytes of
Mailbox 0   */
#define CAN_MBX0C *(volatile unsigned int *)0x7206    /* CAN 2 of 8 Bytes of
Mailbox 0   */
#define CAN_MBX0D *(volatile unsigned int *)0x7207    /* CAN 2 of 8 Bytes of
Mailbox 0   */
#define CAN_MSGID1L     *(volatile unsigned int *)0x7208    /* CAN Message ID
for MB1 , low           */
#define CAN_MSGID1H     *(volatile unsigned int *)0x7209    /* CAN Message ID
for MB1 , high    */
#define CAN_MSGCTRL1 *(volatile unsigned int *)0x720A /* CAN Message Control
Field 1           */
#define CAN_MBX1A *(volatile unsigned int *)0x720C    /* CAN 2 of 8 Bytes of
Mailbox 1   */
#define CAN_MBX1B *(volatile unsigned int *)0x720D    /* CAN 2 of 8 Bytes of
Mailbox 1   */
#define CAN_MBX1C *(volatile unsigned int *)0x720E    /* CAN 2 of 8 Bytes of
Mailbox 1   */
#define CAN_MBX1D *(volatile unsigned int *)0x720F    /* CAN 2 of 8 Bytes of
Mailbox 1   */
#define CAN_MSGID2L     *(volatile unsigned int *)0x7210    /* CAN Message ID
for MB2 , low           */
#define CAN_MSGID2H     *(volatile unsigned int *)0x7211    /* CAN Message ID
for MB2 , high    */
#define CAN_MSGCTRL2 *(volatile unsigned int *)0x7212 /* CAN Message Control
Field 2           */
#define CAN_MBX2A *(volatile unsigned int *)0x7214    /* CAN 2 of 8 Bytes of
Mailbox 2   */
#define CAN_MBX2B *(volatile unsigned int *)0x7215    /* CAN 2 of 8 Bytes of
Mailbox 2   */
#define CAN_MBX2C *(volatile unsigned int *)0x7216    /* CAN 2 of 8 Bytes of
Mailbox 2   */
#define CAN_MBX2D *(volatile unsigned int *)0x7217    /* CAN 2 of 8 Bytes of
Mailbox 2   */
```

```
#define CAN_MSGID3L     *(volatile unsigned int *)0x7218    /* CAN Message ID
for MB3 , low           */
#define CAN_MSGID3H     *(volatile unsigned int *)0x7219    /* CAN Message ID
for MB3 , high    */
#define CAN_MSGCTRL3 *(volatile unsigned int *)0x721A /* CAN Message Control
Field 3           */
#define CAN_MBX3A *(volatile unsigned int *)0x721C    /* CAN 2 of 8 Bytes of
Mailbox 3   */
#define CAN_MBX3B *(volatile unsigned int *)0x721D    /* CAN 2 of 8 Bytes of
Mailbox 3   */
#define CAN_MBX3C *(volatile unsigned int *)0x721E    /* CAN 2 of 8 Bytes of
Mailbox 3   */
#define CAN_MBX3D *(volatile unsigned int *)0x721F    /* CAN 2 of 8 Bytes of
Mailbox 3   */
#define CAN_MSGID4L     *(volatile unsigned int *)0x7220    /* CAN Message ID
for MB4 , low           */
#define CAN_MSGID4H     *(volatile unsigned int *)0x7221    /* CAN Message ID
for MB4 , high    */
#define CAN_MSGCTRL4 *(volatile unsigned int *)0x7222 /* CAN Message Control
Field 4           */
#define CAN_MBX4A *(volatile unsigned int *)0x7224    /* CAN 2 of 8 Bytes of
Mailbox 4   */
#define CAN_MBX4B *(volatile unsigned int *)0x7225    /* CAN 2 of 8 Bytes of
Mailbox 4   */
#define CAN_MBX4C *(volatile unsigned int *)0x7226    /* CAN 2 of 8 Bytes of
Mailbox 4   */
#define CAN_MBX4D *(volatile unsigned int *)0x7227    /* CAN 2 of 8 Bytes of
Mailbox 4   */
#define CAN_MSGID5L     *(volatile unsigned int *)0x7228    /* CAN Message ID
for MB5 , low           */
#define CAN_MSGID5H     *(volatile unsigned int *)0x7229    /* CAN Message ID
for MB5 , high    */
#define CAN_MSGCTRL5 *(volatile unsigned int *)0x722A /* CAN Message Control
Field 5           */
#define CAN_MBX5A *(volatile unsigned int *)0x722C    /* CAN 2 of 8 Bytes of
Mailbox 5   */
#define CAN_MBX5B *(volatile unsigned int *)0x722D    /* CAN 2 of 8 Bytes of
Mailbox 5   */
#define CAN_MBX5C *(volatile unsigned int *)0x722E    /* CAN 2 of 8 Bytes of
Mailbox 5   */
#define CAN_MBX5D *(volatile unsigned int *)0x722F    /* CAN 2 of 8 Bytes of
Mailbox 5   */
#define GPTCONA         *(volatile unsigned int *)0x7400    /* GP Timer
Control Register EVA    */
#define T1CNT           *(volatile unsigned int *)0x7401    /* GP Timer 1
Counter Register        */
#define T1CMPR          *(volatile unsigned int *)0x7402    /* GP Timer 1
Compare Register        */
#define T1PR            *(volatile unsigned int *)0x7403    /* GP Timer 1
Period Register         */
#define T1CON           *(volatile unsigned int *)0x7404    /* GP Timer 1
Control Register        */
#define T2CNT           *(volatile unsigned int *)0x7405    /* GP Timer 2
Counter Register        */
#define T2CMPR          *(volatile unsigned int *)0x7406    /* GP Timer 2
Compare Register        */
```

```
#define T2PR            *(volatile unsigned int *)0x7407    /* GP Timer 2
Period Register         */
#define T2CON           *(volatile unsigned int *)0x7408    /* GP Timer 2
Control Register        */
#define COMCONA         *(volatile unsigned int *)0x7411    /* Compare
Control Register  EVA         */
#define ACTRA           *(volatile unsigned int *)0x7413    /* Full-Compare
Action Register         */
#define DBTCONA         *(volatile unsigned int *)0x7415    /* Dead-Band
Timer Control Register */
#define CMPR1           *(volatile unsigned int *)0x7417    /* Full Compare
Unit Compare Register 1 */
#define CMPR2           *(volatile unsigned int *)0x7418    /* Full Compare
Unit Compare Register 2 */
#define CMPR3           *(volatile unsigned int *)0x7419    /* Full Compare
Unit Compare Register 3 */
#define CAPCONA         *(volatile unsigned int *)0x7420    /* Capture
Control Register  EVA               */
#define CAPFIFOA  *(volatile unsigned int *)0x7422    /* Capture FIFO Status
Register EVA            */
#define CAP1FIFO  *(volatile unsigned int *)0x7423    /* Two-Level-Deep
Capture FIFO Stack 1    */
#define CAP2FIFO  *(volatile unsigned int *)0x7424    /* Two-Level-Deep
Capture FIFO Stack 2    */
#define CAP3FIFO  *(volatile unsigned int *)0x7425    /* Two-Level-Deep
Capture FIFO Stack 3    */
#define CAP1FBOT  *(volatile unsigned int *)0x7427    /* Capture 1 Bottom
Stack Register     */
#define CAP2FBOT  *(volatile unsigned int *)0x7428    /* Capture 2 Bottom
Stack Register     */
#define CAP3FBOT  *(volatile unsigned int *)0x7429    /* Capture 3 Bottom
Stack Register     */
#define EVAIMRA         *(volatile unsigned int *)0x742C    /* EVA Interrupt
Mask Register A   */
#define EVAIMRB         *(volatile unsigned int *)0x742D    /* EVA Interrupt
Mask Register B   */
#define EVAIMRC         *(volatile unsigned int *)0x742E    /* EVA Interrupt
Mask Register C   */
#define EVAIFRA         *(volatile unsigned int *)0x742F    /* EVA Interrupt
Flag Register A   */
#define EVAIFRB         *(volatile unsigned int *)0x7430    /* EVA Interrupt
Flag Register B   */
#define EVAIFRC         *(volatile unsigned int *)0x7431    /* EVA Interrupt
Flag Register C   */
#define GPTCONB         *(volatile unsigned int *)0x7500    /* GP Timer
Control Register EVB    */
#define T3CNT           *(volatile unsigned int *)0x7501    /* GP Timer 3
Counter Register        */
#define T3CMPR          *(volatile unsigned int *)0x7502    /* GP Timer 3
Compare Register        */
#define T3PR            *(volatile unsigned int *)0x7503    /* GP Timer 3
Period Register         */
#define T3CON           *(volatile unsigned int *)0x7504    /* GP Timer 3
Control Register        */
#define T4CNT           *(volatile unsigned int *)0x7505    /* GP Timer 4
Counter Register        */
```

```
#define T4CMPR          *(volatile unsigned int *)0x7506    /* GP Timer 4
Compare Register        */
#define T4PR            *(volatile unsigned int *)0x7507    /* GP Timer 4
Period Register         */
#define T4CON           *(volatile unsigned int *)0x7508    /* GP Timer 4
Control Register        */
#define COMCONB         *(volatile unsigned int *)0x7511    /* Compare
Control Register  EVB         */
#define ACTRB           *(volatile unsigned int *)0x7513    /* Full-Compare
Action Register EVB     */
#define DBTCONB         *(volatile unsigned int *)0x7515    /* Dead-Band
Timer Control Register EVB */
#define CMPR4           *(volatile unsigned int *)0x7517    /* Full Compare
Unit Compare Register 4 */
#define CMPR5           *(volatile unsigned int *)0x7518    /* Full Compare
Unit Compare Register 5 */
#define CMPR6           *(volatile unsigned int *)0x7519    /* Full Compare
Unit Compare Register 6 */
#define CAPCONB         *(volatile unsigned int *)0x7520    /* Capture
Control Register EVB                */
#define CAPFIFOB  *(volatile unsigned int *)0x7522    /* Capture FIFO Status
Register EVB            */
#define CAP4FIFO  *(volatile unsigned int *)0x7523    /* Two-Level-Deep
Capture FIFO Stack 4 */
#define CAP5FIFO  *(volatile unsigned int *)0x7524    /* Two-Level-Deep
Capture FIFO Stack 5 */
#define CAP6FIFO  *(volatile unsigned int *)0x7525    /* Two-Level-Deep
Capture FIFO Stack 6 */
#define CAP4FBOT  *(volatile unsigned int *)0x7527    /* Capture 4 Bottom
Stack Register    */
#define CAP5FBOT  *(volatile unsigned int *)0x7528    /* Capture 5 Bottom
Stack Register    */
#define CAP6FBOT  *(volatile unsigned int *)0x7529    /* Capture 6 Bottom
Stack Register    */
#define EVBIMRA         *(volatile unsigned int *)0x752C    /* EVB Interrupt
Mask Register A   */
#define EVBIMRB         *(volatile unsigned int *)0x752D    /* EVB Interrupt
Mask Register B   */
#define EVBIMRC         *(volatile unsigned int *)0x752E    /* EVB Interrupt
Mask Register C   */
#define EVBIFRA         *(volatile unsigned int *)0x752F    /* EVB Interrupt
Flag Register A   */
#define EVBIFRB         *(volatile unsigned int *)0x7530    /* EVB Interrupt
Flag Register B   */
#define EVBIFRC         *(volatile unsigned int *)0x7531    /* EVB Interrupt
Flag Register C   */
```

VITA

Chakradhar Rao Medavarapu was born in Hyderabad, India, on November 2, 1982, to Rajeswar Rao Medavarapu and Radhika Medavarapu. He finished his schooling in Alpha Public School, Nalgonda, Andhra Pradesh, India, in 1998. He graduated from Little Flower Junior College, Hyderabad, in 2000 completing his Intermediate Education (high school) with majors in mathematics, physics and chemistry (MPC). He graduated with distinction in 2004 from University of Madras, India, where he did his Bachelor of Engineering in Computer Science and Engineering. He joined the Computer Science Department at Louisiana State University to pursue his Master of Science in Systems Science degree in January 2005 (Spring 2005). Currently he is working with Dr. Gerald Baumgartner in the Embedded Systems Group. His research interests include embedded systems, computer interfaces and distributed systems. His hobbies include watching movies, singing, dancing, cooking and calligraphy. He has a younger sister, Navatha PriyaDarshini Medavarapu who is currently working as a Software Engineer in Infosys, Hyderabad, India.