In compliance with the
Canadian Privacy Legislation
some supporting forms
may have been removed from
this dissertation.

While these forms may be included
in the document page count,
their removal does not represent
any loss of content from the dissertation.

DSP Hardware Surrogate for the HLA

by

Lidan Li

A thesis submitted to the

Faculty of Graduate Studies and Research in partial fulfillment of the requirements for

the degree of

Master of Applied Science

In Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

September 2003

© Li, Lidan 2003

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services

Acquisitions et
services bibliographiques

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

ISBN: 0-612-88881-9

Our file Notre référence

ISBN: 0-612-88881-9

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

Canada

Abstract

With the increasing design complexity of embedded systems, the traditional technique of independent hardware and software design is now being challenged. This research focuses on developing a convenient simulation environment that supports hardware and software component interactions in the early design stage. A High Level Architecture (HLA) based system framework is developed that allows the incremental incorporation of hardware subsystems into a software simulation so that hardware-in-the-loop execution is achieved. The proposed framework is verified by a case study that illustrates the integration of two hardware components, i.e. the EVM DSP board and the soundcard, into the simulation of a spectrum analyzer.

This research addresses several limitations found in the current state-of-the-art in HWIL simulation. A two-stage methodology is developed to gain the advantage of HWIL simulation. In the first stage, a pure software simulation is developed. The second stage incrementally extends the software simulation by replacing software modules with hardware modules. Additional goals of the methodology include the reuse of a generic approach for different hardware subsystems, and to allow software module code from the first stage to be reused with the hardware prototype in the second stage.

Acknowledgements

I would like to thank my supervisor, Professor Trevor Pearce for his guidance, assistance, encouragement and suggestion. In particular, this thesis would not have been possible without his attention to detail, enlightening discussions and helping me through the research process.

I would also like to thank my husband, Chunlei Zhang for his encouragement and support throughout my thesis.

Table of Contents

Abstract	iii
List of Acronyms	viii
List of Figures	x
List of Tables	xii
Chapter 1: Introduction	1
Chapter 2: Background Information	7
2.1 Embedded System Development	7
2.2 High Level Architecture (HLA)	9
2.2.1 Run Time Infrastructure (RTI)	10
2.2.2 RTI Management Areas	12
2.3 The TMS320C6x Evaluation Module (EVM) DSP Board	14
2.3.1 Real Time Data Exchange (RTDX)	15
2.4 Spectrum Analysis	17
2.4.1 Fast Fourier Transform (FFT) Algorithm	18
Chapter 3: Hardware-in-the-loop Simulation	20
3.1 HWIL Simulation for Classical Control Systems	20
3.1.1 Hardware and Software Components in HWIL	21
3.2 Underwater Robot Vehicle Research	23
3.2.1 HWIL Simulation	23
3.2.2 Implementation in HLA	25
3.3 Distributed Co-simulation Backbone (DCB)	26

3.4 Generic Toolbox for Interoperable System – GTI6	28
Chapter 4: The Thesis	31
Chapter 5: The HLA-based Approach	34
5.1 Layer 1, RTI Services and Management	35
5.2 Layer 2, Federates' Roles and Responsibilities	36
5.3 Layer 3, Surrogate Implementation	37
5.4 Layer 4, Plug-in Software or Hardware	38
Chapter 6: Case Study: Spectrum Analyzer	40
6.1 Stage 1 Implementation	40
6.1.1 High Level Architecture	40
6.1.2 Object/Interaction Class Design	42
6.1.3 Simulation Activity Diagram	43
6.1.4 SignalGenerator Federate	45
6.1.5 DSP Federate	49
6.1.6 Display Federate	54
6.2 Stage 2 Implementation	57
6.2.1 Incorporating the DSP Board into the Simulation	58
6.2.1.1 RTDX Communication between the Host and Target	59
6.2.1.2 RTDX Data Flow	60
6.2.1.3 RTDXchannel Class	61
6.2.1.4 RTDX DSP Board Side Program	64
6.2.2 Incorporating the Soundcard into the Simulation	66
6.2.2.1 Soundcard	66

6.2.2.2 HWIL High Level Architecture	68
6.2.2.3 SoundSurrogate Federate Design	69
Chapter 7: Case Study: Verification and Performance Analysis	73
7.1 Test Set Up	73
7.2 Verification	74
7.2.1 Test Case 1: Federation Communication	74
7.2.2 Test Case 2: Comparison of the FFT Calculation Results	77
7.2.3 Test Case 3: Message Loss in the Simulation	78
7.3 Performance Analysis	80
7.3.1 Sampling Rate Limitation	80
7.3.2 RTI Latency and Bandwidth	81
7.3.3 Hardware Bottle Neck	83
Chapter 8: Conclusions	86
8.1 Conclusions	86
8.2 Future Research Directions	87
References	89

List of Acronyms

API	Application Programming Interface
CCS	Code Composer Studio
COTS	Commercial-off-the-shelf
DCB	Distributed Co-simulation Backbone
DFT	Discrete Fourier Transform
DoD	Department of Defense
DSP	Digital Signal Processor
EVM	Evaluation Module
FedExec	The Federation Executive
FFT	Fast Fourier Transform
FOM	Federation Object Model
GTI6	The Generic Toolbox for Interoperable Systems
HLA	High Level Architecture
HWIL	Hardware-in-the-loop
LibRTI	The RTI Library
MFC	Microsoft Foundation Class
MOM	Management Object Model
OMT	Object Model Template
PLC	Programmable Logic Controller
RTDX	Real Time Data Exchange

RTI	Run Time Infrastructure
RtiExec	The RTI Executive
SOM	Simulation Object Model
TBC	Test Bus Controller

List of Figures

Figure 1-1	High Level Architecture of the Approach	4
Figure 2-1	Workflow in Traditional Development Cycle	8
Figure 2-2	Logical View of RTI Components	11
Figure 2-3	RTI and Federate Code Responsibilities	12
Figure 2-4	Attribute Update Protocol	14
Figure 2-5	RTDX Block Diagram	15
Figure 2-6	Target-to-Host Communication	16
Figure 2-7	Host-to-Target Communication	17
Figure 2-8	FFT Block Diagram	19
Figure 3-1	HWIL System Architecture	21
Figure 3-2	Hardware-in-the-loop Simulation Approach	24
Figure 3-3	Architecture of the DCB	27
Figure 3-4	GTI6 Architecture	29
Figure 5-1	System Level Block Diagram	35
Figure 5-2	Abstract Interface Block Diagram	37
Figure 5-3	Code Reuse for Programmable Hardware	39
Figure 6-1	High Level Architecture of Stage 1	41
Figure 6-2	Object Class	42
Figure 6-3	Simulation Activity Diagram	44
Figure 6-4	Class Relationship of SignalGenerator Federate	46

Figure 6-5	Wave Generation	46
Figure 6-6	Pseudo Code for signalGeneratorFederate Class	48
Figure 6-7	Class Relationship of DSP Federate	49
Figure 6-8	Pseudo Code for DSPFederate Class	53
Figure 6-9	Screen Shot of the Display Federate	55
Figure 6-10	Class Relationship of Display Federate	56
Figure 6-11	Pseudo Code for MagDisplayFederate Class	57
Figure 6-12	High Level Architecture of HWIL Simulation	58
Figure 6-13	Data Flow between DSPsurrogate and DSP board	61
Figure 6-14	Data Flow State Chart	62
Figure 6-15	Pseudo Code for processWave Function	63
Figure 6-16	Pseudo Code on Target Board	65
Figure 6-17	Soundcard A/D Conversion	67
Figure 6-18	Soundcard Data Gathering	68
Figure 6-19	High Level Architecture after Stage Two	69
Figure 6-20	Screen Shot of SoundSurrogate Federate	70
Figure 6-21	Class Relationship of SoundSurrogate Federate	70
Figure 7-1	Distributed Simulation Platform	74
Figure 7-2	Results for the Pure Software Simulation	76
Figure 7-3	Results from Both Cases	78
Figure 7-4	Message Loss Test Results	79
Figure 7-5	Comparison of the Performance	84

List of Tables

Table 2-1	The Activities in the RTI Management Areas	13
Table 6-1	Object Class Publishing and Subscription	43
Table 6-2	Interaction Table	43
Table 6-3	RTDX Functions	60
Table 7-1	Sample Rate at Different Level	81
Table 7-2	Object Update Rate under Different Sample Rate	82
Table 7-3	Object Update Rate	83

Chapter 1 Introduction

Gradually software engineering has become a well-established industry with standards and quality controls. Methodologies are evolving to dramatically increase the "productivity" of developers in making products of better quality, substantially cheaper and easier to maintain [1]. For instance, the increasing complexity of embedded system design combined with reductions in the time-to-market window is revolutionized the embedded system design process. The traditional design techniques of independent hardware and software design are now being challenged by development cycle and quality assurance concerns when integrating heterogeneous models and applications to create complex systems.

In the early stage of traditional design techniques for embedded systems, software and hardware are usually developed and validated in separate design processes. A key factor in the length of the resulting development cycle is the availability of the hardware. In an extreme case, significant development delay and development deadlock may occur. For example, the software development team may wait for the hardware prototypes to become available, while the hardware development team requires the software environment to verify and test the hardware prototypes. Moreover, serious problems that may require significant hardware rework or software workarounds may not be discovered until later during the system integration phase [2]. Delays in addressing integration issues can also delay the product development cycle.

With the expansion of embedded system markets, both hardware and software components are becoming more complex [3]. Products often incorporate pre-developed, complex, application-specific blocks, e.g. DSP, ASIC and FPGA, together with large software components that are matched to the hardware design. The ambitious scale of the products makes integrating and verifying the hardware and software components more difficult. To deal with development demands, simulation is being used increasingly to supply early feedback on design and implementation decisions. Hardware engineers have embraced simulation as a cornerstone in hardware development methods [4]. Software engineers are exploring the use of high-level simulators that can execute abstract design and implementation specifications [5]. Despite the acceptance of simulation as a valuable tool in the separate development of hardware and software components, methodologies and tools for the co-development of hardware and software components are still topics for ongoing research [6].

The problems outlined above stimulate the need for a development environment that supports both simulation and a surrogate harness for hardware sub-systems. A surrogate harness is an interface that allows the simulation to interact with software simulation models or hardware prototypes. The surrogate allows the user to integrate and test the functionality of software and hardware components earlier in the development process. Such an approach has the potential to reduce overall development costs, shorten the product development cycle, and reduce the time-to-market period.

Software reuse can also play an important role in the development cycle by reducing the time to market and improving software quality and reliability [7]. To support a development process that improves software reuse, an integrated reusable framework is desired. Reuse within the framework can be at different levels, such as software code reuse or software architecture reuse. A reusable framework can be an effective structure for incorporating different software or hardware components. It provides generic and important functionality for application systems in order to encourage reuse and to avoid re-discovery, re-invention and re-implementation.

The main objective of this research is to develop a simulation environment that allows both software and hardware components to be integrated and tested at an earlier development stage. Additional objectives are to develop a reusable generic framework for integrating hardware subsystems, and to promote the reuse of software code between software simulations and hardware prototypes. In order to meet these objectives, the High Level Architecture (HLA) is chosen to be the simulation environment. The HLA is a general-purpose framework developed under the Department of Defense (DoD) for modeling and simulation [8]. The HLA is a standard for distributed simulation and integration. It supports reuse and interoperability across a large number of different types of simulations and their components. The general structure of the interface of the HLA makes it a good choice to promote reuse and to achieve interoperability among heterogeneous simulation components. In addition, the HLA supports reliable transmission of the messages among the simulators as well as receive ordered messaging scheme that suits the need

of this research.

The results of this research are an HLA-based simulation environment and a development method in which hardware subsystems are introduced incrementally into a software simulation. Ideally, this hardware-in-the-loop (HWIL) simulation approach will allow the functionality of the software and hardware to be integrated and tested earlier in the development of embedded systems.

Figure 1-1: High Level Architecture of the Approach

The approach is achieved in the two stages outlined in Figure 1-1. In stage one, an HLA simulation harness interacts with a software module that simulates the behavior of the hardware. The Run Time Infrastructure (RTI) allows the other HLA-compliant federates to exercise the functionality of the software module. In stage two, the simulation harness and software module are replaced with the surrogate harness connected to the real hardware prototype. The RTI now allows the other HLA-compliant federates to interact with the real hardware, and HWIL execution is achieved.

The remainder of this document is structured as follows.

Chapter 2 provides a brief overview of background information pertaining to development methods and the tools used in this research. The HLA is introduced as the target distributed simulation environment. The research results are illustrated in Chapter 6 using a spectrum analysis application. In the case study, the EVM board from Texas Instruments (TI) is used as the hardware prototype. To lay the foundation for the case study, background information on the EVM board and the Fast Fourier Transform (FFT) algorithm are presented in Chapter 2.

Chapter 3 reviews the main streams of the research related to HWIL simulation. It discusses the classical HWIL approach used in control system simulations, and some uses of the HLA as a distributed platform to incorporate HWIL simulation. Chapter 3 also provides a critical analysis of these research approaches.

Chapter 4 states the thesis, the research objectives, the proposed solution, the thesis scope and a summary of the contributions made by the research.

The solution is described in Chapters 5 and 6. Chapter 5 details the proposed HLA-based framework and the two-stage development method. The solution is discussed in high-level terms as well as a four-layer structure that clearly identifies the backbone (RTI), the simulator (federates), the HLA surrogate harnesses and the software/hardware modules. The layered structure facilitates the reuse of stage one artifacts in the second stage. Chapter 6 demonstrates the solution by implementing a spectrum analysis case study. Initially, a pure software simulation of a spectrum

analyzer is developed. Hardware prototypes (the EVM DSP board that performs FFT calculation, and the soundcard that generates the digital wave samples) are gradually introduced into the simulation to replace their corresponding software modules. The description of each software/hardware module and the development of the overall system are given in detail.

Chapter 7 verifies the correctness of the system via a set of test cases and analyzes the performance of the simulation by varying system loads (by varying the sampling rates). The effectiveness of using the HLA as a target environment is examined, as well as the success of using the proposed development method.

Chapter 8 states the conclusions of the research, and suggests some future research directions.

Chapter 2 Background Information

This chapter reviews the traditional development method for embedded systems, and presents a current method of co-design for embedded systems. The following components fill various roles in the research. The High Level Architecture (HLA) [8] is a standard distributed simulation framework that supports the interoperation of heterogeneous simulation components. The HLA is used as the target simulation framework for incorporating hardware and software modules. The TMS320C6x Evaluation Module (EVM) board [9] from Texas Instruments (TI) is a general-purpose platform for developing embedded Digital Signal Processor (DSP) applications. A key feature of the board is the Real Time Data Exchange (RTDX) technology, which provides easy data communication between the board and host applications. The EVM board is used as a hardware component to demonstrate the proposed two-stage development method. Spectral Analysis [10] is a typical DSP application, and is the system application used in the case study to illustrate the development method. In order to make this document self-contained, a brief introduction of each of these components is provided in the following sections.

2.1 Embedded System Development

With the increasing complexity of the embedded system design and fiercely competitive markets, the traditional development method for embedded systems is

being challenged to increase functionality while reducing the time-to-market [11].

Figure 2-1 shows the typical workflow in a traditional development cycle.

Figure 2-1: Workflow in Traditional Development Cycle [11]

When the system specification is developed, the hardware team and software team separately develop their own parts. Typically, software is developed after a hardware design has begun to stabilize, and prototypes are available for integrating the software and hardware. The limited observability of hardware operations while the software executes, and the inability of either team to control all of the elements of the design can have a negative impact on the development cycle. Furthermore, the need for hardware and/or software rework due to system design flaws is often delayed until the integration phase. The delay in discovering these flaws ripples back into the development cycle to cause further product development delays.

With advances in hardware modeling and simulation, a new design methodology using hardware/software co-design is evolving to reduce the embedded system development cycle and the time-to-market [12]. Hardware/software co-design is the cooperative

design of hardware and software components by enabling the tradeoffs among options in one domain to affect design decisions in the other. Compared with the traditional development cycle, the co-design method supports hardware and software development concurrently. When the hardware prototypes are not available, software can still be developed using a simulated model of the hardware. The software can be exercised at an earlier stage, and therefore earlier feedback is available regarding system and component design. Ideally, when a hardware prototype becomes available, it could replace the simulated model without requiring extensive rework of the software. The co-design method can shorten development time by encouraging concurrent development of hardware and software components, starting the integration stage earlier, and (ideally) simplifying the integration stage.

2.2 High Level Architecture (HLA)

The Department of Defense (DoD) introduced the High Level Architecture (HLA) as a standard framework to support simulations composed of heterogeneous components [8]. The HLA has its roots in defense programs, but was designated the IEEE-1516 standard in 2000 [13]. It proposes rules and mechanisms for the interoperability of a distributed, heterogeneous simulator. The objectives of the HLA are to facilitate interoperability among simulations, and to promote reuse of simulations and their components. It specifies the general structure of the interfaces between simulation components without making specific demands on the implementation of each simulation. In HLA terminology, a *federate* is an HLA-compliant simulation

component, and a *federation* is a simulation composed of a set of interacting federates.

The HLA allows federates to interoperate by sharing attributes (data) and/or interaction (events), and also by sharing a common notion of time.

The HLA consists of three components:

- **Federation rules:** The HLA has ten rules, which must be obeyed if a federate or a federation is to be regarded as HLA compliant. The HLA rules are divided into two groups consisting of five rules for HLA federations and five rules for HLA federates.
- **Interface specification:** The interface specification defines a standard application programming interface (API) for the Run-Time Infrastructure (RTI). The RTI is a middleware product that enables the interoperation and distribution of federates. The RTI services include a set of “callback” functions that each federate must implement to receive communications from the RTI. The RTI software conforms to the HLA interface specification but is not itself part of the specification.
- **Object Model Template (OMT) [14]:** The OMT provides a standard for documenting HLA object model information. The OMT defines the simulation object model (SOM), the federation object model (FOM) and the management object model (MOM). The SOM describes the shared objects used by a individual federate. The FOM describes the shared objects used within a federation (i.e. the union of all SOMs). The MOM identifies objects used to manage the federation.

2.2.1 Run Time Infrastructure (RTI)

The RTI middleware provides common services to federates and federations. It implements the HLA interface specification and provides the software services necessary to support an HLA-compliant simulation. The following is a high level logical view of an executing HLA federation.

Figure 2-2: Logical View of RTI Components [8]

All of the components shown in Figure 2-2 are part of a single federation, with the exception of the RtiExec. The shaded portions are provided by the RTI, while the remainder are federate specific. The Federation Executive (FedExec) manages the federation. It allows federates to join and to resign from the federation, and facilitates data exchange among participating federates. The RTI executive (RtiExec) is a global process that manages the creation and destruction of multiple federations. The RTI library (libRTI) makes RTI services available to each federate.

Federates use libRTI to invoke HLA services. Figure 2-3 illustrates the RTI and federate code responsibilities. Within libRTI, the class RTIambassador bundles the

services provided by the RTI. All requests made by a federate on the RTI invoke an RTIambassador method. The abstract class FederateAmbassador identifies the callback functions each federate is obliged to provide. While both the RTIambassador and the FederateAmbassador classes are a part of libRTI, the federate must implement the functionality declared in FederateAmbassador.

Figure 2-3: RTI and Federate Code Responsibilities [8]

2.2.2 RTI Management Areas

The HLA Interface Specification partitions the services provided by the RTI into the following six management areas: federation management, declaration management, object management, ownership management, time management, and data distribution management. In this research, only the first three areas are employed. Table 2-1 provides a synopsis of the actions employed.

Management Area	Activities Described	Action Synopsis
Federation Management	Manages federation execution. Initializes name space, transportation, routing spaces, etc.	<ul style="list-style-type: none"> - Creation - Joining - Resigning - Deleting
Declaration Management	Specify data types a federate will send and receive.	<ul style="list-style-type: none"> - Publication - Subscription
Object Management	Creates, modifies, and deletes objects and interactions. Facilitates object registration and distribution. Coordinates attribute updates among federates.	<ul style="list-style-type: none"> - Register Object - Update Attribute - Send Interaction - Delete Object

Table 2-1: The Activities in the RTI Management Areas

Federation management is responsible for federation creation and deletion. It also provides the utilities for a federate to join and to resign from the federation. Declaration management allows a federate to publish and subscribe shared classes among all federates within the federation. Each federate is responsible for identifying its publication and subscription interests to the RTI using the RTIambassador services `subscribeObjectClassAttributes()` and `publishObjectClass()`. Information is exchanged using objects, which are instances of shared classes. Object management deals with object registration and deletion, attribute update and interactions send/receive. In other words, it manages the message flow among the federates. In order to communicate the values of the attributes among the interested federates, an object instance must first be declared and registered to the RTI. Figure 2-4 shows the protocol for attribute value update. When the publishing federate invokes `updateAttributeValues()`, the RTI knows which federates subscribe to the attribute. The RTI calls the subscribing federate (via the `reflectAttributeValues()` callback) to

communicate the updated values.

Figure 2-4: Attribute Update Protocol

2.3 The TMS320C6x Evaluation Module (EVM) DSP Board

The TMS320C6x evaluation module (EVM) is a low-cost, general-purpose platform for the development, analysis, and testing of 'C6x digital signal processor (DSP) algorithms and applications [9]. The EVM can be used to evaluate whether the processor and programmed algorithms meet application requirements. The EVM is bundled with Code Composer Studio (CCS) [15], Windows 95 and NT drivers, host PC and DSP software APIs, example applications with source code, and various utility applications.

The EVM used in this research contains the 'C6201 fixed-point DSP, which operates at up to 1600 MIPS with a CPU clock rate of 200 MHz. Also, the EVM provides embedded JTAG emulation, using an onboard test bus controller (TBC), as well as a header to support an XDS510 JTAG emulator. Of particular importance to this

research, the embedded JTAG emulation supports the real time data exchange technology between the target EVM board and the host PC.

2.3.1 Real Time Data Exchange (RTDX)

The Real Time Data Exchange (RTDX) technology allows the transfer of data between a host and target device without interfering with the target application [16].

Figure 2-5 shows the system configuration to utilize the RTDX.

Figure 2-5: RTDX Block Diagram

In Figure 2-5, the RTDX consists of both target and host library components. The target application makes function calls to the target library's API in order to send or receive data. The RTDX software library transfers data to/from the host in the background while the target application is running. On the host platform, the RTDX host library operates in conjunction with CCS. In order to send or receive data between the host and target board, an input or output channel must be declared for both sides. Display and analysis tools can communicate with the RTDX host library via an easy-to-use COM API to obtain target data and/or to send data to the target

application. Designers may use their choice of standard software packages to retrieve, analyze, and/or display data in various formats such as Microsoft's Visual C++, Visual basic, and even Excel.

Figure 2-6 shows the target-to-host RTDX communication. An output channel must be configured on the target before data can be transferred. Data may then be written to the output channel using API routines defined in the RTDX library. This data is immediately recorded into a target buffer in the RTDX target library and then sent to the host through the JTAG interface. The RTDX host library receives data from the JTAG interface and records it into either a memory buffer or an RTDX log file. The transfer of data through the JTAG to host is done autonomously without interrupting the target's processor. The data recorded by the host can then be collected by the host application and processed in a meaningful way.

Figure 2-6: Target-to-Host Communication

Figure 2-7 shows the host-to-target RTDX communication. For the target to receive data from the host, an input channel on the target side must be declared. The target requests data from the input channel using RTDX library service. This request is

recorded into the target buffer and then sent to the host through the JTAG interface. Data to be sent to the target is written to a memory buffer inside the RTDX host library through a COM interface. When the RTDX host library receives a read request from the target application, the data in the host buffer is sent to the target through the JTAG interface. The data is then written to the requested location on the target in real time. The host then notifies the RTDX target library when the operation is completed.

Figure 2-7: Host-to-Target Communication

2.4 Spectrum Analysis

Fourier transform based signal processing, which is often simply termed “spectral analysis”, is used extensively for modern-day engineering tasks, including speech recognition, vibration analysis, and biomedical engineering [10]. Spectrum analysis is a typical DSP application that can make use of a DSP chip. The objective of spectrum analysis is to determine the frequency, amplitude and phase of the sinusoidal components present in a signal. It simplifies the performance of some mathematical

operations on a signal, and quantifies the signal in the frequency domain. The heart of the spectrum analysis is a frequency domain analysis technique known as the Discrete Fourier Transform (DFT), and, in particular, the Fast Fourier Transform (FFT) derivative.

2.4.1 Fast Fourier Transform (FFT) Algorithm

The DFT is simply a mapping of one ordered set of N complex numbers to a different ordered set – the former conveying time domain information which is the sampled digital signal, the latter is the frequency domain information. The DFT formula is as follows [17]:

$$X(k) = \sum_{n=0}^{N-1} x(n)W_{N/2}^{nk} \quad W_{N/2}^{nk} = e^{\frac{-j2\pi nk}{N/2}}$$

Where $x(n)$ represents the input sampled signal and $X(k)/N$ is the magnitude of the spectral line at the frequency $k \cdot (\text{sampling frequency})/N$. N is the number of samples taken per DFT, k and n are indices valid between 0 and N-1. The complex exponential $W_{N/2}^{nk}$ is known as a twiddle factor, which can be represented in rectangular format as the sum of a cosine and imaginary sine value.

The Fast Fourier Transform (FFT) is a significantly less computationally intensive method of evaluating the DFT, and thus particularly attractive for ‘real-time’ spectral analysis using DSP technology. Figure 2-8 shows the FFT block diagram. The *Recombine Algebra* shown in the diagram is used to ensure the resulting magnitudes are ordered by linearly increasing frequency. The FFT algorithm breaks up the original

N point samples into two ($N/2$) sequences. The decimation process can be repeated (i.e. by breaking down the $N/2$ point DFTs into $N/4$ point DFTs, and then to $N/8$ point DFTs, etc.) until a series of two point DFTs is reached. This is known as a radix-2 decimation-in-time FFT.

Figure 2-8: FFT Block Diagram

The computation of an N point DFT requires $N^2/2$ complex additions and multiplications. The FFT reduces the complexity from N^2 to $N\log_2(N)$. For 1024 point samples, using direct translation of the DFT formula would result in a program that requires over 1 million complex multiplications, while the FFT would only require approximately 10,000 complex multiplications.

Chapter 3 Hardware-in-the-loop Simulation

This section reviews the state-of-the-art for hardware-in-the-loop (HWIL) simulation relevant to this research [18,19,20,21,22,23]. Since there has been numerous works done in this research direction, it is impossible to give complete details. However, focus will be given to the following representative streams of research:

- The concept of HWIL design and implementation in classical control system development.
- Development and integration of a distributed HWIL system based around the HLA. The interoperability and synchronization issues are discussed.
- A generic architecture called the Distributed Co-simulation Backbone (DCB).
- A Generic Toolbox for Interoperable Systems (GTI6), which includes the implementation of a distributed environment based on the HLA, and the interface to HWIL and surrogate facilities.

3.1 HWIL Simulation for Classical Control Systems

There are many groups conducting research in HWIL simulation for classical control systems [24,25,26]. In his paper, Grega describes the concept of HWIL design system, which he implemented as an experimental part of a control engineering course [20]. The HWIL methodology integrates a simulation model of the process being controlled with the target control hardware. The simulation model provides all the processing signals in real-time. The signals are then converted by D/A modules and supplied to

the controller as voltages. Analog control signals are produced by the controller and supplied via A/D converters back to the simulation model. The purpose of this configuration is to create an environment for the hardware component that behaves as closely as possible to the real environment that would be encountered by a deployed control system.

3.1.1 Hardware and Software Components in HWIL

For a classical control experiment, the control unit requires a direct connection with the plant. By using the HWIL approach, the plant is substituted by a simulated plant model that provides all input signal to the real target, i.e. the programmable logic controller (PLC). The proposed system consists of a PLC controller, a PC-type computer where the simulation model is located, and another PC used for PLC code development. Figure 3-1 shows the resulting system architecture.

Figure 3-1: HWIL System Architecture [20]

As indicated in Figure 3-1, the hardware component in the loop is the PLC controller, and the simulated plant software components include:

- Tools for modeling and simulation (MATLAB/SIMULINK)
- Library of I/O drivers: a set of pre-defined blocks, representing the functions implemented in hardware
- Code generation software for automatic building of real time models
- User-interface for visualization of data and on-line tuning of the parameters.

The simulation model of the plant is developed using MATLAB and the model is supplied with I/O drivers to connect to the I/O board. The simulation model provides plant environment signals and feeds them through a D/A module to the PLC. The PLC reacts to the environment signal information by generating analog control signals, which are supplied back to the simulated plant via A/D converters. This communication pattern allows the PLC to sense and control the plant. The advantages of incorporating the HWIL approach into the control engineering course include:

- Low-cost of the laboratory experiment. By incorporating the HWIL method in the laboratory, students can set up the experiment in the laboratory instead of requiring a real plant. This reduces the costs of the lab and increases the effectiveness of teaching.
- Hardware algorithms and design methods, which are used on the PLC can be quickly verified by the simulated plant models. This, along with the rapid prototyping path, speeds up the development and verification cycle of the

control systems.

In the above research, the HWIL method is focused on the operation of a hardware device with its inputs and outputs coupled to a simulation model instead of the real working environment. If such system could also provide a surrogate for a simulation of the PLC, then the functionality of the PLC can be tested without the physical presence of the hardware and the control algorithms can be verified before the control unit is built. This could further reduce the costs, enhance early correction of design flaws and improve teaching efficiency (i.e. students could examine their control algorithms without having to wait for the shared PLC to become available).

3.2 Underwater Robot Vehicle Research

Distributed HWIL simulations normally involve interoperability and synchronization considerations. Through the development of an open HLA framework for integrating underwater robot vehicle subsystems, Lane et al [21] demonstrated that HWIL simulation is beneficial for the interoperability of simulated and real subsystems. This section will give a brief review of the HWIL approach and the implementation framework used in the underwater robot vehicle research.

3.2.1 HWIL Simulation

Lane's methodology for distributed HWIL simulation is to interoperably combine the simulations with real hardware. Figure 3-2 reveals a logical view of the relationship between simulated and physical subsystems. On the left, there are simulated sensors,

actuators and an environment for initial testing and validation. In the centre, prototypes of physical sensors, actuators and a test environment are used for initial HWIL subsystem testing. On the right, the real sensors/actuators, as deployed with the final system. A common communication spine infrastructure is employed throughout the development process.

Figure 3-2: Hardware-in-the-loop Simulation Approach [21]

Initially, pure simulation components are employed. As the real subsystems are integrated and tested, the simulation components are removed gradually until the real robot is realized. To seek the interoperability of the HWIL simulation, two cases are conducted to test the results of the simulations and hardware combination. In case one, real sensors and simulated actuators are selected. In order for the real sensor to work consistently with the simulated actuator, additional hardware is used to provide sufficient motion bandwidth and additional software is implemented to integrate the sensors with the communication spine. On the contrary, case two selects the simulated

sensor and real actuator. Once again, additional data flows are used to incorporate the real actuator with the communication spine. The results of both cases show that the interoperability of the simulated hardware and real hardware can be achieved.

3.2.2 Implementation in HLA

The framework for integrating real and simulated subsystems is developed within a distributed computing environment based on the communications software spine. The vehicle communication spine in Figure 3-2 is implemented using the HLA RTI provided by the DoD. It includes six federates (pilot's interface, bottom station, propulsion, sensing, vehicle dynamics, simulation management) to coordinate the interactions among the simulated/real hardware subsystems and to manage the overall system behavior and simulation progress. The framework allows the simulations and real hardware to interoperate in a simulated environment.

By using the HWIL method in the distributed simulation, the real hardware and the simulated subsystems can be tested and integrated in the early design stage. However, to achieve the complete interoperability of simulations and real hardware subsystems, additional data flow that connects the hardware and the simulation environment is required, and must be modified every time a simulation is replaced by hardware and vice versa. This adds complexity to the simulation and slows down the development cycle. If there were a standard interface between the hardware and the simulation environment that provides generic functions to connect the hardware and the simulations, it would be more convenient to add/remove hardware without having to

expend extra effort in accommodating the additional data flow. Furthermore, although the research achieved interoperability of software simulations and hardware to a certain level, the author admits that complete interoperability of simulated and real subsystems for the robot HWIL is difficult in practice due to timing synchronization issues, and he questions the long term validity of the simulation.

3.3 Distributed Co-simulation Backbone (DCB)

The DCB is a distributed simulation architecture [22] that can be seen as a simulation-specific coordination layer for supporting distributed co-simulation. It is an experimental extension of the HLA standard, in which the interface specification, data exchange and synchronization management areas of the RTI have been extended to make the service more flexible. The DCB uses the concept of a gateway to handle the complexity of the interface specification. The gateways are implemented as part of an ambassador. In addition to bridging the simulation and the DCB, the ambassadors also translate data formats, according to the destination of the data sent through the DCB.

Figure 3-3: Architecture of the DCB [22]

As indicated in Figure 3-3, two ambassadors are associated with each simulator. The DCB ambassador handles the communication from/to the DCB and it is similar to the combination of the RTI ambassador and the federate ambassador in the HLA. The simulator's ambassador is developed as a gateway to the DCB ambassador and it handles the needs of the interface specifications. Both ambassadors must be developed when a new simulator is integrated into a co-simulation environment. The DCB is a general-purpose infrastructure that is independent from the integration of new elements into a federation. The integration of a new simulator imposes requirements that must be met by simulators in terms of interface specification, data management, and synchronization.

The DCB can support transparent cooperation between models generated by a general-purpose discrete system simulator (such as SIMOO [27]). In order to validate the DCB architecture, a federation for the co-simulation of electronic embedded systems is currently being implemented. Also, a supporting environment for the DCB

development methodology is being built.

Although the DCB is an attractive approach to support heterogeneous software simulators working in a distributed simulation environment, the DCB and supporting development methods are still under development. DCB has only been applied to heterogeneous software simulators and has not included real hardware subsystems to create HWIL simulations. Furthermore, the inherent extensions to the HLA result in an environment, which is not compliant with the HLA standard.

3.4 Generic Toolbox for Interoperable System (GTI6)

The GTI6 toolbox developed by EADS Launch Vehicles (EADS LV) [23] in the Automated Transfer Vehicle (ATV) project focuses on employing a surrogate to achieve HWIL simulation, and on validating the simulation results. According to the HWIL methodology used, the ATV project can be tested in a closed loop with real-time performance. The purpose of GTI6 is to support distributed tasks by providing the necessary tools to:

- Allow collaborative working between distant teams;
- Manage shared engineering databases;
- Interconnect distributed simulation facilities in real time.

The GTI6 architecture contains nine generic components that can be used (entirely or partially) for distributed and interoperable systems.

Figure 3-4: GTI6 Architecture [23]

As indicated in Figure 3-4, the Middleware (MW) is a core element in the GTI6 architecture as it lies among the Simulation Framework, the Surrogate, the Supervisor, the Synchronization, and the Network and Security services. The main component in the Middleware is the HLA RTI. Another important component is the Surrogate (SRG). The Surrogate supports the geographical distribution of simulation facilities in a closed loop and in real time with real HWIL. The purpose of a Surrogate is to give a local node exactly the same capabilities as those of the centralized simulation, by hiding the distribution. Some key functionality carried out by the surrogate is as follows:

- Replace remote hardware equipment locally;
- Communicate with remote components;
- Respect the same hardware and software interface and behaviour inherent to the replaced component;

- Give to the simulator or to the operator the impression of communicating directly with one another in a centralized manner, by compensating possible perturbations involved by the distribution.

By using the Surrogate, the geographical distribution of facilities can be tested in a closed loop and in real time. The Surrogate can replace the remote hardware component working with a local software simulation, and it behaves exactly the same as the replaced hardware component. The research has focused on preserving a standard interface to both the simulation and the actual hardware, and has successfully achieved plug and play interoperability. However, the use of HWIL is only restricted to the testing and simulation of remote hardware. Furthermore, the authors do not discuss methodologies to develop the plug and play modules. They simply state that the modules can be interchanged.

There are also other similar state-of-the-art researches in the related field that are not discussed one by one; however, the above given examples are among the most closely related to this thesis research and are good representatives of the trend of the HWIL research directions and approaches.

Chapter 4 The Thesis

During the development and integration of embedded system components, HWIL simulation enables testing at a much earlier stage in the development cycle. It allows the simulated and real hardware subsystems to interoperate with each other. By the time the system integration phase begins, the embedded system has been tested much more thoroughly than it would have been in the traditional approach.

The current state-of-the-art in HWIL simulation reviewed in Chapter 3 has several limitations. The classical control system approach does not address the development of components in the control system. The work of Lane et al [21] has achieved some success; however, the integration of HWIL seems to consistently require extra development to deal with additional data flows. The DCB is not HLA compatible, has not been shown to support HWIL, and is still an ongoing research project. The GTI6 architecture has achieved plug and play interoperability for HWIL simulation; however, a methodology for construction plug and play modules has not been described. This study will address the above limitations.

The thesis of this research is that the advantages of HWIL simulation can be gained by a methodology in which a software simulation is incrementally extended by replacing software modules with hardware prototype. The additional goals of the methodology include enhanced traceability in the development process during the work product evolution, the reuse of a generic approach for different hardware

subsystems, and to allow software module code to be reused in the hardware prototype.

The proposed methodology to achieve the goals of the thesis is a two-stage method based on the HLA. Stage one has a pure software simulation, with a module that simulates the hardware behaviour. Stage two replaces the software module with real hardware to achieve HWIL simulation. The details of the two-stage method are presented in Chapter 5. To illustrate the application of the methodology, a spectrum analysis case study is presented in Chapter 6. Chapter 7 provides a critical analysis of the performance and the overall results.

The scope of the thesis is limited to:

- Defining a simulation framework using the RTI Next Generation 1.3 (RTI-NG 1.3) implementation according to the HLA interface specification version 1.3. The C++ programming language is used in this implementation.
- Demonstrating the development method using the spectrum analysis case study. For this case study, two hardware subsystems are included: the EVM DSP board and the soundcard. Only the RTDX technology is used in the CCS API. The target and the host program are all written in the C language. The soundcard is a general device residing in the PC. Only the Analog-to-Digital (A/D) technology of the soundcard is used in this case study.

The contributions of this research are:

- An HLA-based simulation method that supports both software and hardware

prototypes interaction. The method is achieved in two stages, and encourages code reuse.

- A harness federate architecture organized into abstract interface layers between the hardware/software modules and the RTI middleware.
- A general framework for accessing compatible products using the Code Composer Studio (CCS) environment and the RTDX technology.
- Achieve plug and play by defining a standard interface for the framework to communicate with both software and hardware modules.
- Examine and prove the effectiveness and simplicity of using such a method.
- Analysis of the performance of the system to verify the effectiveness of HWIL simulation based on the HLA environment and the EVM DSP boards using RTDX technology.

Chapter 5 The HLA-based Approach

This chapter presents the system level HLA-based framework and the two-stage development method. The federates in each of the stages are structured into four layers as described below.

The proposed method for incorporating hardware into software simulation and verifying the functionality of the hardware is a major component in this research. Design is done in a unified framework so as to prejudice neither hardware nor software implementation. The development is divided into two stages. Stage one involves pure software simulation that simulates the hardware behavior. Stage two replaces the software module with real hardware to achieve HWIL simulation. The Figure 5-1 shows the design block diagram that explains the overall structure and the two-stage approach.

Figure 5-1: System Level Block Diagram

Figure 5-1 (a) reveals the first stage, a pure software simulation. Figure 5-1 (b) represents the hardware replacing the software to achieve hardware-hardware communication. Both stages contain a four-layer structure. Layer 1 is the Run-Time Infrastructure (RTI), which represents the communication layer between different federates. Layer 2 represents federate specific code that automates the services provided by the RTI within a federation. Layer 3 contains a surrogate that is used to connect federates with the software module or hardware environment. Layer 4 represents different independent software modules or hardware subsystems. The following sections introduce the functionality of each layer in detail.

5.1 Layer 1, RTI Services and Management

The layer 1 represents the communication services provided by the HLA RTI. The HLA RTI is a Commercial-off-the-shelf (COTS) component. There are six management areas in the RTI as described in the section 2.2.2. These RTI interface services are used by federates to send information (data or events) to other federates to serve the communication needs within a federation.

5.2 Layer 2, Federates' Roles and Responsibilities

Layer 2 is the federate interface layer. This layer provides the functional services in the federate to exchange information with the RTI. The lifecycle activities for the federate in a federation are as follows:

1. Create or join a federation.
2. Initialize the object/interaction and register them with the federation.
3. Declare the data/event it will send or publish to the federation. Also, declare data/event that it is interested in receiving or subscribing to/from other federates.
4. Resign from the federation when the simulation is over.

Although this is a generic layer needed by each federate to communicate with the RTI, the information to exchange (i.e. the specific objects/interactions to publish/subscribe) is application specific and is different from one federate to another. During the two-stage (Figure 5-1) development, the federate interface layer remains unchanged. In

other words, this layer must be developed in stage 1, and is then reused in stage 2.

5.3 Layer 3, Surrogate Implementation

Interface design is critical in achieving easy plug and play of hardware/software modules in the framework. From the federate's point of view, the interface to its hardware or software modules are the same. In other words, the calls that a federate makes to its hardware does not differ from those calls the same federate makes to the corresponding software components. The functional classes in the abstract interface are modularized to achieve this purpose. Figure 5-2 shows the abstract interface class block diagram.

Figure 5-2: Abstract Interface Block Diagram

The surrogate provides three modularized sections: interface initialization, interface access function and interface destruction. The surrogate is organized in such a way in order to accommodate the start up/close down and access procedures for the hardware, and to keep the interface to Layer 2 consistent. The interface initialization section takes care of the initialization of the software module or the hardware subsystem with

required information (e.g. data size, data types). The interface access function section implements the various function calls that can be made by the calling applications and obtains the results returned by the hardware/software modules. The interface destruction allows the release of the software/hardware modules after use. During migration from Figure 5-1 (a) to Figure 5-1 (b), the surrogate interfaces to the calling applications remain the same to achieve easy plug and play interchange of the software and hardware in Layer 4; however, the internal implementation of each of the interface functions will be different. When interchanging Layer 4 components, the implementations of the Layer 3 interface functions must also be interchanged.

5.4 Layer 4, Plug-in Software or Hardware

According to the development method, at the first stage, the hardware functionality is simulated by a software module. In the second stage, the software module is removed and the real hardware is used. Ideally, both stage realize the same functionality. For programmable hardware components, an optimal development strategy will support code reuse across the two stages. Figure 5-3 illustrates potential code reuse between the software module and the real hardware. On the left side, the software module is used. The functional class is divided into two parts: interface related and hardware function related. On the right side, the real hardware is deployed. The functions implemented by the hardware are also divided into two parts: hardware communication and hardware functions. By classifying the functions in the software/hardware, the code that implements the hardware related functions in the software module is reused in the

(programmable) hardware. When the software module is developed, part of the code for the hardware part is also embedded. This reduces the hardware development time, avoids re-implementation of similar functionality and serves the reuse objective of this research. By reusing the software code, the simulated software module precisely presents the functionality of the hardware, which makes the simulation results more reliable.

Figure 5-3: Code Reuse for Programmable Hardware

Chapter 6 Case Study: Spectrum Analyzer

This chapter describes the detailed implementation of a spectrum analyzer case study, which illustrates the framework and development method proposed in chapter 5. It verifies the two-stage approach shown in Figure 5-1.

6.1 Stage 1 Implementation

This section describes the first stage of the development method: the pure software simulation in the HLA framework (Figure 5-1 a). It starts with the introduction of the high level architectural design of the spectrum analyzer, followed by the description of the object and interaction classes used (i.e. the FOM), and the detailed implementation of each federate in the simulation.

6.1.1 High Level Architecture

The case study simulates the operation of an FFT spectrum analyzer with its input signal provided by a signal generator and its output frequency components displayed as a graph. According to the operation of the spectrum analyzer, three federates can be built in the “spectrum analyzer” federation. They are: signalGenerator, DSP and display federates. Figure 6-1 shows the high level architecture of the spectrum analyzer federation.

Figure 6-1: High Level Architecture of Stage 1

The functionality of each federate is described as follows:

- **SignalGenerator:** The signalGenerator federate is responsible for generating digital samples of waveforms. In stage one, the software version only generates sine and cosine waves. The default waveform is a sine wave. The wave's frequency and sampling rate are predefined at 1kHz and 8kHz, respectively. Any signal component frequencies exceeding the Nyquist limit of 4 kHz (half the sampling rate) will result in aliasing.
- **DSP:** The DSP federate is responsible for carrying out the radix-2 FFT algorithm. The FFT algorithm imposes the constraint that the number of input samples must be equal to a power of 2. The sampling points are set to 1024 throughout the case study and the maximum sampling rate is set to 44.1K (the maximum sampling rate a regular PC soundcard can handle).
- **Display:** The display federate is a passive federate. It receives the magnitude values from the DSP federate and updates them on a graphic window. The display

graphic view is in two dimensions, the x-axis represents the frequency and the y-axis represents the magnitude.

6.1.2 Object/Interaction Class Design

This section identifies the object classes and the interaction classes used by the federation that are defined in the FOM. Within the Spectrum Analyzer federation, there are two object classes defined: Wave and Magnitude. In this document, capital “Wave” and “Magnitude” refer to the classes while lower case “wave” and “magnitude” are the object instances of the “Wave” and “Magnitude” classes.

The object classes defined in the FOM can be seen in Figure 6-2. The Wave object class belongs to the singalGenerator federate and it modifies the attributes of the wave. The DSP federate owns the Magnitude object class that allows the magnitude attributes to be modified by the DSP federate.

Figure 6-2: Object Class

Table 6-1 lists the published and subscribed attributes for the Wave and Magnitude

classes.

Object Class	Attributes/Data Type	signalGenerator Federate	DSP Federate	Display Federate
Wave	Points/short	Publish	Subscribe	N/A
	Frequency/short	Publish	Subscribe	N/A
	SampleRate/short	Publish	Subscribe	N/A
	Samples/short	Publish	Subscribe	N/A
Magnitude	MagResult/float	N/A	Publish	Subscribe

Table 6-1: Object Class Publishing and Subscription

Information within a federation can also be transferred between federates via interactions. In the Spectrum Analyzer federation, the SimulationEnds interaction is used to indicate that the federation should terminate. Table 6-2 lists the published and subscribed parameters for the interaction class defined in the FOM.

Interactions/Type	signalGenerator Federate	DSP Federate	Display Federate
SimulationEnds/bool	Publish	Subscribe	Subscribe

Table 6-2: Interaction Table

SimulationEnds := TRUE when the federate should stop and resign from the federation. This is the only interaction used in the entire simulation and is invoked from the signalGenerator federate to all other federates to complete the simulation.

6.1.3 Simulation Activity Diagram

Figure 6-3 is an activity diagram that illustrates the sequence and interactions of the Spectrum Analyzer federation.

Figure 6-3: Simulation Activity Diagram

As indicated in the above diagram, there are three federate life cycles. Each of the cycles contains the following steps:

Step 1: If the federation does not exist, the federate (whichever starts first) will create a new one and add itself to this federation. If a federation already exists, then the federate joins that federation. The function `CreateAndJoinFederation()` is used by the federate to attempt to create the federation and then join it.

Step 2: After joining the federation, the federate is initialized. Initialization includes publishing/subscribing object and interaction classes and performing object/interaction registrations. The function `PublishAndSubscribe()` is used to achieve this functionality.

Step 1 and 2 are the creation and initialization life cycle of the simulation/federation execution. These two steps are the same for all federates.

Step 3: Following step 2, each federate enters a loop containing federate-specific processing. During this part of the life cycle, federates communicate with each other by interacting with instances of the object classes defined in the FOM. Detailed descriptions of the federates, their layers, roles and responsibilities are presented in the following sections.

Step 4: The simulation loop iterates until the termination flag is set to true. All federates then resign from the federation, delete object instances and attempt to destroy the federation execution. This is accomplished by a call to `EndSimulation()`. This step ends the entire simulation.

The following sections present the detailed descriptions of each federate involved in the simulation.

6.1.4 SignalGenerator Federate

As described in the previous section, this federate is responsible for generating the sine wave for further processing by the DSP federate. The signalGenerator federate includes four classes: `signalGeneratorFederate` class, `Generator` class, `Wave` class and `GenFedAmbassador` class. The class relationship is shown in Figure 6-4.

Figure 6-4: Class Relationship of the SignalGenerator Federate

- Generator class:** This class generates 1024 data points when the `generateWave` function is called. The newly created data points are placed in an object passed into the class. The `generateWave` function itself represents the functionality of the software module (Layer 4 of Figure 5-1). The surrogate interface (Layer 3) consists of three functions: `Generator::Generator(SizeOfWave);`, `Generator::generateWave(wave);` and `Generator::destroy()`. The code fragment shown in Figure 6-5 reveals how the data points are generated.

```

Generator::generateWave(Wave::points) {
    M_PI = 3.1415927;
    theta = 2.0 * M_PI * Frequency / SampleRate;
    loop from 0 to Samples-1 {
        Wave::points[index] = sin(index*theta);
        index increment;
    }
    sleep(Samples/SampleRate);
}
  
```

Figure 6-5 Wave Generation

This function simulates the sine and cosine wave according to the math function:

$\sin(\square)$ or $\cos(\square)$, where $\square = 2\pi * \text{Frequency} / \text{SampleRate}$

The frequency of the sine or cosine wave is set to 1kHz and the sampling rate is 8kHz. In the case study, the FFT algorithm processes 1024 sample points at a time, therefore the generator is initialized to handle a wave size of 1024, and `generateWave()` generates 1024 sample points per invocation. The period of a 1kHz sine wave is 1ms. Since the signal is sampled 1024 times at intervals of 1/8 KHz (0.125 ms), there are 128 complete cycles of the wave encoded in the sample points generated by each call to `generateWave()`. The `sleep()` function is called to give the federate periodic behaviour (with period equal to $\text{Samples} / \text{SampleRate}$). The sleep interval is calculated based on the sampling rate and the number of sample points taken, so the simulation loop generates and updates new sample points at a rate equal to the desired sampling rate.

- **Wave class:** The Wave class is created for inter-federate communications. The member function `sendChanges()` allows changes made in wave data to be reflected in the DSP federates, and to accomplish communication from one federate to the other. The function `sendChanges()` is invoked after the sample points have been generated and updated in the wave's Points attribute. Every time this function is called, the newly updated Points are reflected to federates who subscribe to this attribute.
- **GenFedAmbassador (Layer 2):** This is the federate ambassador of the

signalGenerator federate, and it provides callback functionality when new messages arrive. For the signalGenerator federate, callback functions are not implemented since this federate does not subscribe to any of the objects or interactions.

- **signalGeneratorFederate class:** This class handles the overall communication requirements needed for the federate. It makes calls to the RTI (Layer 1) services and coordinates the other three classes to play the roles of the signalGenerator federate. Figure 6-6 shows the specific function carried out by the signalGeneratorFederate class in the simulation loop.

```
void SignalGeneratorFederate::runSim() {  
    createAndJoinFederation();  
    publishAndSubscribe();  
    while ( !terminate ) {  
        Generator::generateWave(Wave->points);  
        Wave->SendChanges();  
    }  
    EndSimualtion();  
}
```

Figure 6-6: Pseudo Code for signalGeneratorFederate Class

The signalGeneratorFederate execution performs federation creation and allows the federate to join the federation (Layer 2, federate interface to communicate with RTI). Then it enters the while loop, where it repeatedly generates a set of wave points and sends them to the DSP federate. When the federation terminates, the

signalGeneratorFederate resigns from the simulation.

6.1.5 DSP Federate

The DSP federate subscribes to the wave object generated by the signalGenerator federate, and processes wave Points using the FFT algorithm. This federate includes six classes: DSPFederate class, Wave class, Magnitude class, RTDXchannel class, FFT class and DSPFedAmbassador class. Their class relationship is shown in Figure 6-7 below.

Figure 6-7: Class Relationship of the DSP Federate

- **Wave and Magnitude classes:** The Wave class is the same one that is used in the signalGenerator federate. The use of this class in two federates is, however, slightly different. The wave object is being subscribed to this federate, in contrast to being published by the signalGenerator federate. The federate ambassador detects the

attribute change and reflects the change via the call back function. The Magnitude class is similar to the Wave class, and has functions such as `sendChange()`. The DSP federate publishes a single instance (object) of the Magnitude class called “magnitude”.

- **DSPFedAmbassador class (Layer 2):** This class acts as the federate ambassador of the DSP federate. This class extends the `RTI::FederateAmbassador` class to provide a callback function interface for the user to implement. The `RTI::FederateAmbassador` is an abstract class with a predefined interface as stated in HLA's programmer guide [8]. In this case study, the following callback services are used and must be implemented.

- *discoverObjectInstance()*: This callback method is called when the wave object instance is registered by the signalGenerator federate. The method provides an object handle, which is used to identify the object for subsequent updates.
- *reflectAttributeValues()*: When the attributes of the wave object are updated by the signalGenerator federate, this method is called and the new attribute values are passed to the DSP (subscribing) federate.
- *receiveInteraction()*: This callback method is called when the “terminate” interaction is received by the federate ambassador. After receiving the interaction, actions are taken to terminate the federate.
- *removeObjectInstance()*: This callback method informs the federate that a previously discovered object no longer exists. When the wave object is deleted

by the singalGenerator federate, this method is called to inform the federate that the wave object no longer exists.

- **RTDXchannel class:** This class provides a uniform interface that ensures no changes are needed to the DSP federate when the hardware module replaces the software module in the second stage of the case study. The interface functions (Layer 3) accessible by the DSP federate include `RTDXchannel()`, `processWave()` and `DestroyRTDX()`.
 - *The `RTDXchannel(waveSize)`:* This function is the class constructor. It initializes the channel with the size of the wave for processing (i.e. the number of sample points for each FFT calculation). The size defaults to 1024 as required by the application. The constructor allocates sufficient memory for data structures and returns a Boolean value indicating whether the initialization has succeeded.
 - *`processWave(wave)`:* This function accepts the array of points to be processed and returns the magnitudes calculated by the FFT algorithm. It first takes the 1024 points and assigns them to an internal data structure. It then passes the information to the FFT class (layer 4), obtains the returned magnitude values and passes the results back.
 - *`DestroyRTDX()`:* This function is the class destructor. It simply closes the RTDX channel and deallocates the memory used.
- **FFT class (Layer 4):** This class implements the FFT algorithm in two steps. Step

one performs the decimation-in-time method, in which the input elements' indices will be reordered. In second step, the process combines the two-point DFT and twiddle multiplication into one processing operation. After the FFT algorithm is performed, within the same routine, the magnitude values are calculated. The entire FFT calculation is carried out in function `calculateFFT()`. Later in the second development stage, when the EVM hardware board is brought into the simulation, this class is replaced by the DSP board and the `calculateFFT()` function will be moved directly onto the DSP board without modification. In other words, when the simulated DSP model has been developed, part of the hardware code has also been developed. This shortens the development time and makes the results more reliable.

- **DSPFederate class:** Similar to the `signalGeneratorFederate` class, the `DSPFederate` class performs federation creation and destruction and acts as a bridge (Layer 2) between the software FFT surrogate and the RTI. Figure 6-8 shows the pseudo code for the `DSPFederate` class in the simulation loop.

```

void DSPFederate::runSim() {
    createAndJoinFederation();
    publishAndSubscribe();
    RTDXChannel(); //initialize FFT software module.
    While ( !terminate ) {
        rtiAmb.tick( 0.01, 0.01 ); //leave room for federate
                                   //ambassador callback function
                                   //to execute

        if (FFT_Performed ) {
            Magnitude->SendChanges();
            FFT_Performed:= false;
        }
    }
    DestroyRTDX();
}

EndSimulation();
}

```

Figure 6-8: Pseudo Code for DSPFederate Class

After joining the federation, the DSPFederate enters a loop, but since callbacks control the periodic behaviour of the process, the function uses the tick service. `rtiAmb.tick()` allows the RTI a chance to execute and respond to requests. When `rtiAmb.tick( 0.01, 0.01 )` is called from the main loop, it allows the callback functions to update the value of the wave object and invokes the FFT functions to perform the FFT calculation. The parameters passed to tick determine the interval of time allocated to tick processing. In this case, control is relinquished to tick for a time not less than 0.01 seconds (the first parameter), nor greater than 0.01 seconds (the second parameter), i.e. exactly 10 msec. During this period of time, the callback routine may update the wave points, process the wave points using FFT

algorithm and set the FFT_Performed flag to be true. After the 0.01 seconds elapsed, the control is returned to the DSP federate. If the wave has been processed, it will update the magnitude results to the display federate. The time set in the tick function does make a difference on the performance and the accuracy of the federate. If the time tick is too long, there might already have been two updates on the wave object and two FFT calculations have been performed. The second magnitude value would then overwrite the first one. When the magnitude value is finally sent from the main loop, the display federate receives only one update instead of two, resulting in message loss. If the time tick is too short, the callback may not have taken place. If no wave is received and no FFT is performed, the tick won't have achieved anything and valuable CPU time is wasted. This does not cause message loss, but slows down the performance of the federate. Therefore, the time tick must be designed carefully according to the simulation needs. In this case study, the maximum sampling rate is 44.1K. As a result, we expect a wave object update every 0.02 sec, and setting the time tick at 0.01 seconds appears to be optimal.

6.1.6 Display Federate

Unlike the other two active federates, the display federate passively waits to receive updates of objects and interactions. It does not send any information to other federates. In order for the results to be viewed and updated easily, this federate is built as Microsoft Foundation Class (MFC) application [28]. It provides a two-dimensional

graphical view. Figure 6-9 shows a visual snapshot of the magnitude display.

Figure 6-9: Screen Shot of the Display Federate.

The x-axis represents the frequency range. The maximum frequency variations depend on the frequency of the incoming wave. The x-axis range is 4 KHz by default, which is half the 8KHz sampling rate. It is divided into 512 increments (half of 1024) with each increment of 7.8 Hz. The y-axis represents the magnitude of the frequency components. The view is a control chart that is implemented as a Microsoft ActiveX control. The application obtains a handle to the chart and is able to update the chart at runtime. The classes involved in this federate are shown in Figure 6-10 and discussed below. The Magnitude class works in a similar way to the Wave class addressed in the earlier section will not be restated again.

Figure 6-10: Class Relationship of the Display Federate

- **DisplayFedAmbassador class (Layer 2):** This class is the federate ambassador implementation of the display federate, and is similar to the federate ambassador in the DSP federate. DisplayFedAmbassador class overwrites discoverObjectInstance(), reflectAttributeValues(), receiveInteraction() and removeObjectInstance() functions. These functions are redefined in a similar way as the DSP federate, except for the subscribed object is magnitude instead of wave.
- **MagDisplayFederate class:** Unlike other federate classes, the MagDisplayFederate class is derived from the MFC CFrameView class [28] in order to obtain a handle to the chart object and for updating the chart view. The main routine is given in Figure 6-11.

```

void MagDisplayFederate::runSim() {
    createAndJoinFederation();
    publishAndSubscribe();
    obtainChartHandle();
    while ( !terminate ) {
        rtiAmb.tick( 0.01, 0.01 ); //leave room for federate
                                   //ambassador callback
                                   //function to execute

        if (mag_updated ) {
            chartHandle.updateChart(Magnitude);
        }
    }
    EndSimulation();
}

```

Figure 6-11: Pseudo Code for MagDisplayFederate Class

Upon application start up, the federate joins the federation, subscribes to the magnitude object, and obtains the handle to the chart control. It then waits for the update of the Magnitude attribute. If new magnitude data is received, the federate simply updates the chart with new magnitude value. The application terminates on the reception of the termination command.

The above concludes the design and implementation of the pure software simulation.

Verification of the simulation will be discussed in Chapter 7.

6.2 Stage 2 Implementation

The second stage gradually incorporates hardware subsystems into the stage 1 software simulation. As a first step, the EVM DSP board is introduced. As a second step, to

further integration of the hardware during development, a hardware soundcard is introduced. The following sections discuss the design and implementation of each step in detail.

6.2.1 Incorporating the DSP Board into the Simulation

The TMS320C6x EVM DSP board (serving as the Layer 4 component) must be introduced into the simulation and the FFT algorithm moved onto the DSP board. The advantages of introducing the DSP board are that the DSP architecture is designed for numerically intensive algorithm, and the stage 1 code for the FFT algorithm can be reused.

The high level architecture of the spectrum analyzer federation with HWIL simulation is shown in Figure 6-12.

Figure 6-12: High Level Architecture of HWIL Simulation

The federation consists of the signalGenerator, DSPsurrogate and display federates. The signalGenerator and display federates remain unchanged from stage 1, as described

in section 6.1. The DSPsurrogate federate, however, replaces the DSP federate used in the stage 1 federation. The FFT algorithm implemented in the DSP federate is now moved to the DSP board. The CCS Integrated Development Environment (IDE) supports the DSP board and RTDX feature. By using the RTDX between the host and the target board, the surrogate federate can communicate with the DSP board to obtain FFT processing.

The DSPsurrogate federate is named differently from the DSP federate to distinguish the HWIL federation from the stage 1 federation. The main changes lie inside the RTDXChannel class. As in the pure software simulation, this class represents Layer 3 in the overall structure, and the interface functions used to obtain FFT processing from the RTDXchannel class remains unchanged. However, since the class now interacts with the DSP board instead of the software module, the implementations of the RTDXChannel() (constructor), processWave() (FFT service) and destroyRTDX() (destructor) functions must perform different tasks than they did in the pure software simulation. The following sections detail the design of each of the functions and the data communications to the target DSP board.

6.2.1.1 RTDX Communication between the Host and Target

The RTDX service provided by the CCS IDE is used to exchange data between the host and the target board. Data communication between the host and target board has already been introduced in section 2.3. Table 4 lists the functions that handle

communications between the host and target.

Function	PC (host)	DSP (target)
Initialize RTDX	m_rtdx->Open m_rtdx->Close where m_rtdx is an IRtdxExpPtr object	RTDX_CreateInputChannel RTDX_CreateOutputChannel RTDX_enableOutput RTDX_enableInput
Read from RTDX	m_rtdx->Read	RTDX_read
Write to RTDX	m_rtdx->Write	RTDX_write

Table 6-3: RTDX Functions

First, the RTDX channel must be initialized on both the host and target before data can be exchanged. An object of type of IRtdxExpPtr is used to manage data exchange. After the channel is initialized, it is ready for exchanging (reading or writing) data. Upon termination of the federation, the RTDX channel must be closed, and the IRtdxExpPtr object released.

6.2.1.2 RTDX Data Flow

Figure 6-13 below summarizes the data flow state between the DSPsurrogate federate and the DSP board.

Figure 6-13: Data Flow between DSPsurrogate and DSP Board

After initializing and joining the federation, the DSPsurrogate federate enters a loop. It waits for the subscribed attributes (points) to be updated by the signalGenerator federate. When acquiring the new points, the DSPsurrogate federate sends these points to the DSP board through the RTDX channel. Once the board finishes processing these points, magnitude values are returned to the DSPsurrogate federate. Then the latter updates the results for the display federate.

6.2.1.3 RTDXchannel Class

Figure 6-14 shows the data flow chart of the communication between the RTDXchannel class and the target application on the DSP board. On the host side, after initialization, the RTDX channel waits for the new sample points from the signalGenerator federate. Once the points data is ready, the write() function sends the data to the target DSP board. On the target board side, the RTDX channel is initialized

then it enters an infinite loop. Within the loop, the RTDX channel tries to read data from the host. If there is no data available, the RTDX channel is blocked. Once data has arrived, it processes the data and sends the results back through the output channel. Once the results are sent back to the host, the data is updated for the display federate.

Figure 6-14: Data Flow State Chart

The detailed descriptions of the RTDX channel functions are as follows:

- **RTDXchannel():** This function includes the RTDX initialization on the host PC.
- **processWave():** This function involves two steps: write wave data to, and then read magnitude data from, the target board. Figure 6-15 shows the section of the code that indicates how the processWave() function communicates with the DSP board in detail.

```
float* RTDXChannel::processWave(short* waveData, int nSize) {
    waveArray = SafeArray.create(waveData);
    if (rtdx not initialized)    //if channel not initialized, no point
                                //in continuing
        return;
    waveArray.initialize(waveData); //place waveData into the safeArray
    Rtdx.Write( waveArray );    // write the waveData to the board
    Rtdx.Flush();                //enforce the board to start processing
                                //immediately
    SafeArrayDestroy(waveArray); //destroy safe array after finish
    float MagData[512]; //allocate memory for magnitude, half size of
                        //the wave
    magArray = SafeArray.create(MagData);
    while(Rtdx.Read(magArray) != success) {} //keep reading until
                                              //success
    return MagData;
}
```

Figure 6-15: Pseudo Code for processWave Function

The function starts with a check on whether the RTDX input channel has been opened. A SafeArray is created and initialized to hold the wave data. The SafeArray is an RTDX specific mechanism for handling safe data exchange between the PC and the DSP board [29]. The next step is to simply write the SafeArray to the board using the write function. The flush function is called to

enforce the board to process the wave data immediately. Once the data is handed over to the board, it is up to the DSP board to carry out the FFT algorithm. On the host side, the next action is to create another SafeArray and prepare for receiving magnitude data from the board. Unlike the write function, the read function is placed in a while loop. The read function call does not block the caller if no data is available, instead, an error is returned. To ensure the correct retrieval of the magnitude data, the read function is constantly being called until the data is available. The retrieved magnitude is then returned to the calling federate to be sent out to the display federate.

- **destroyRTDX():** This function is used to release the RTDX channel and destroy the RTDX handle. This function is called only when the federate has decided to resign from the federation.

6.2.1.4 RTDX DSP Board Side Program

The target program running on the DSP board is shown in Figure 6-16.

```

RTDX_CreateInputChannel(ichan);    //input channel, accepts data from
                                   //the host
RTDX_CreateOutputChannel(ochan);   //output channel, write data to
                                   //the host

float* CalculateFFT(short* wave, float* mag, int size);

void main()
{
    RTDX_enableInput(&ichan);        // Step 1
    RTDX_enableOutput(&ochan);

    While (!terminate)              // Step 2
    {
        RTDX_read( &ichan, points, sizeof(points) );
        CalculateFFT( points, mag, 1024 );
        RTDX_write(&ochan, mag, 512);
    }

    RTDX_disableInput(&ichan);        //Step 3
    RTDX_disableOutput(&ochan);
}

```

Figure 6-16: Pseudo Code on the Target Board

The main procedure follows the three steps described in Figure 6-16. Step one initializes and enables the channels. Step two consists of a loop, in which the procedure waits for data to be transferred from the host, performs the FFT calculation, and then returns the results back to the host. Unlike the host RTDX read function, the RTDX read function in the target board blocks until valid data is received, therefore, there is no need for a nested polling loop. Step three terminates the program by disable the channels. Because both the host PC simulation and the target board are written in C/C++, the code of the implementation of the FFT is exactly the same as in the software simulation. The stage 1 source code for the FFT algorithm described in section 6.1.5 is

reused on the target board.

6.2.2 Incorporating the Soundcard into the Simulation

The hardware soundcard is introduced into the simulation to show further integration of hardware during development and to achieve hardware-to-hardware data transfer. The soundcard replaces the signal generator function in the pure software signalGenerator federate, and allows sampling of sound that is input through a microphone. The following sections discuss the detailed implementation of incorporating the soundcard hardware.

6.2.2.1 Soundcard

The soundcard inside a PC typically supports the following basic functions: Synthesizer, MIDI interface, Analog-to-digital conversion for the recording (A/D), and Digital-to-analog conversion during playback (D/A). In this case study, the signal generation in the stage 1 federation is replaced by the direct conversion of analog sound signals from a microphone. The analog-to-digital conversion (A/D) function of the soundcard is employed to convert analog sound signals to digital data for further processing.

Figure 6-17 Soundcard A/D Conversion

Figure 6-17 illustrates how the A/D conversion works. The acoustic waves are input through the microphone and transferred to soundcard, the soundcard then transforms the analog signal into series of digital samples in its A/D converter.

Digital sampling can be done in various qualities including 8 or 16 bit resolution; 11, 22 or 44 KHz (kilohertz) sampling rate; and stereo or mono channels. The quality of the A/D converter is measured in terms of the sampling rate and resolution. The higher the sampling rate, the better the quality becomes, but it also increases the quantity of samples taken. The resolution refers to how much data is recorded in each sample. In this case study, the input is sampled at 16 bit resolution, 44 KHz sampling rate and mono channel. Therefore, an array of 1024 samples will occupy 2 Kbytes of memory. The obstacle of retrieving data from the soundcard is overcome by using Visual C++ Microsoft Foundation Class (MFC) waveform-audio interface functionality that allows easy control over the soundcard. Various implementations of controlling the soundcard are available [30]. To simplify the implementation, existing code to control the

soundcard [31] has been modified to fit our needs. Figure 6-18 shows the basic mechanism of gathering the soundcard data.

Figure 6-18: Soundcard Data Gathering

The main procedure [31] initializes all the parameters of the soundcard, allocates buffers for storing samples, creates an EVENT object and starts the sound thread. The thread procedure is used to signal that a buffer has been filled and that the sound system is ready for a new acquisition.

6.2.2.2 HWIL High Level Architecture

Figure 6-19 shows the high level architecture for this step. A soundcard inside the host PC is used to generate signals for the SoundSurrogate federate. In this way, the user can talk through a microphone, and the soundcard can change these analog signals into the digital samples. Through the RTI, these digital samples will be sent to the DSP board,

and after FFT processing, the frequency components will be sent for display.

Figure 6-19: High Level Architecture after Stage Two

6.2.2.3 SoundSurrogate Federate Design

There is only one-way traffic between the soundcard and the SoundSurrogate federate. The soundcard constantly feeds digitized samples to the federate for communicating to the DSPsurrogate federate. To visualize the waves, the SoundSurrogate federate includes a graphic interface that continuously redraws the waves received from the microphone. Figure 6-20 shows the screen shot of the SoundSurrogate federate display for an arbitrary input.

Figure 6-20: Screen Shot of SoundSurrogate Federate

The SoundSurrogate federate includes the Generator class, Wave class, sndFedAmbassador class, and SoundSurrogateFederate class as shown in Figure 6-21. The functionality of these classes including their attributes and methods are given in sequence.

Figure 6-21: Class Relationship of SoundSurrogate Federate

SoundSurrogateFederate class is named differently from the signalGenerateFederate to

distinguish the HWIL federate from the stage 1 federate. The main changes lie inside the Generator class. The Generator class represents Layer 3 in the overall structure and the interface used to obtain the wave generation is remain unchanged; however, to interact with the soundcard, the implementations of Generator() and destroy() functions must now perform tasks to access the hardware soundcard than it did in the stage 1 simulation. The following section details the design of the constructor, destructor and the generateWave() function of the new Generator class.

The Generator() function (i.e. the constructor of the Generator class) uses openMic() function to enable the microphone and initializes the soundcard for wave generation. The destroy() function calls closeMic() to disable the microphone and releases the soundcard. Inside the generateWave() function, a thread is started. The thread blocks the generateWave() function and waits for the soundcard sampling buffer to be filled. When the thread signals that the buffer is filled, it unblocks the generateWave() function and allows the function to accesses the soundcard sampling buffer to store information into the wave object. Once the wave object is filled, the generateWave() function returns. The SoundSurrogateFederate runSim() function (refer to Figure 6-6) receives the filled wave object and sends it to the DSP federate for further processing. To test the simulation with different sampling rates, the generateWave() function is designed such that it can be controlled dynamically to generate samples at rates of 22.05, 11.025, 5.5 and 2.75KHz.

The above discussion concludes the design of incorporating two hardware subsystems

into the simulation to achieve the HWIL execution. The correctness of the HWIL simulation will be verified in chapter 7.

Chapter 7 Case Study: Verification and Performance Analysis

This chapter presents testing results of the Spectrum Analyzer case study described in Chapter 6. The correctness of the implementation is verified by different test cases. The performance of the simulation is also analyzed to evaluate the effectiveness of the distributed simulation framework.

7.1 Test Set Up

Figure 7-1 reveals the distributed simulation system used for testing. The HLA platform is executed above Microsoft WinNT V4.0. Each federate is executed on a dedicated PC connected to a TCP/IP intranet. All three computers are configured with 128 MB RAM and a Pentium-III CPU with clock speed at 450 MHz. The computer executing the DSP/DSPsurrogate federates also is equipped with an internal EVM6x board (TMS320C6x01 EVM) from Texas Instruments.

Figure 7-1: Distributed Simulation Platform

The following sections describe the verification and performance testing of the simulation system. To permit the collection of test data, federates were instrumented to output messages to their runtime consoles.

7.2 Verification

To verify the correctness of the simulation system in the two development stages, the following tests were conducted:

- (a) Federation communication;
- (b) Comparison of the results from software simulation and HWIL simulation;
- (c) Determination if messages were lost in case (a) and (b).

7.2.1 Test Case 1: Federation Communication

The purpose of this test is to ensure that the communications among the three federates

are correct. The design of the federation allows the start up of the federates in any sequence; however, sequences such as display->DSP->Generator or DSP->display->Generator make more sense for testing purpose since the data flow won't happen until the signalGenerator federate is started. By starting up the signalGenerator first, before the DSP and display could be ready, the waves generated will be lost. Figure 7-2 shows the recorded test results from each federate. The display and DSP federates started up by creating and joining federation, they then blocked and waited for signals from the signalGenerator federate. The signalGenerator federate later on joined the federation and then updated the wave points. The DSP federate soon discovered the object and processed the points. Shortly thereafter, the display federate displayed the magnitude results on the screen. The communication among these three federates is correct, as the attribute is published, subscribed and discovered as expected.

(a) signalGenerator federate console:

```
FED_Gen: CREATING FEDERATION EXECUTION
FED_Gen: SUCCESSFUL CREATE FEDERATION EXECUTION
FED_Gen: SUCCESSFUL JOINED FEDERATION EXECUTION
The following commands can be used to generate the Wave:
Press 's' to generate the sine Wave
Press 'c' to generate the cosine Wave
Press 'q' to quit the application
```

(b) DSP federate console:

```
FED_DSP: CREATING FEDERATION EXECUTION
FED_DSP: CREATION OF FEDERATION EXECUTION SUCCESSFUL
FED_DSP: ATTEMPTING TO JOIN SoundDemo
FED_DSP: SUCCESSFULLY JOINED SoundDemo
FED_DSP: Discovered object 196609
```

(c) Display federate chart:

This is a snapshot of the display window with 1KHz sine wave. As expected, there is only one sinusoidal component, and this component is at the frequency of 1KHz.

Figure 7-2: Results from the Pure Software Simulation.

7.2.2 Test Case 2: Comparison of the FFT Calculation Results

To evaluate the correctness of the stage 1 (pure software) simulation in terms of representing the corresponding hardware module, the results obtained from both simulations (stage 1 and stage 2 with the hardware DSP board) are compared. The signalGenerator federate generated the same 2 KHz sine wave data for both systems. The pure simulation is tested first. The signalGenerator federate generated a 2KHz sine wave for the DSP federate. For verification purposes, the display federate was instrumented to print the magnitude results on the console as seen in Figure 7-3. There are two columns in Figure 7-3 (a), the left column represents the index of the magnitude in the value buffer, and the right column represents the magnitude value. According to the figure, the only significant magnitude value is shown on the 255th index, which is half of the maximum frequency (4KHz) that can be tested. This confirms that the spectral analysis of the input sine wave is as expected. The hardware was then incorporated into the stage 2 simulation for testing. The configuration of the signalGenerator and Display federate remain unchanged, and the DSP federate was replaced by the DSPsurrogate federate, and the DSP board was used to calculate the FFT results. The results are shown in Figure 7-3 (b).

Software FFT	DSP Board FFT
0 0.000003	0 0.000003
1 0.000007	1 0.000016
2 0.000003	2 0.000003
3 0.000004	3 0.000017
...	...
253 0.000004	253 0.000004
254 0.000004	254 0.000018
255 64.00000	255 64.00000
256 0.000057	256 0.000047
257 0.000029	257 0.000019
...	...
510 0.000019	510 0.000022
511 0.000014	511 0.000006

(a)
(b)

Figure 7-3: Results from Both Cases

The corresponding magnitude results from both cases are very close. The maximum difference between corresponding values was found to be about 0.001%. This is due to the differences in the data representation and the compilers (C++ for software version vs. C on the DSP board). This can be ignored when compared to the inaccuracies of 0.003% i.e. 1 LSB (least significant bit) in typical 16-bit A/D converters [32]. This verifies that the FFT algorithm is performed consistently in both cases.

7.2.3 Test Case 3: Message Loss in the Simulation

This test ensures that no messages were lost during transfer in the simulation. A simple method is used to examine the delivery of messages sent through the DSP federate. Two counters were introduced to keep track of how many updates were received, and how many were processed and sent to the display federate. The counts are displayed in the DSP federate console. A third count was created to keep track of the number of updates processed in the last second to see if it matches the sampling rate. It should be noted that

the system is designed to handle up to a 22.05 KHz sampling rate. The system performance depends largely on the CPU speed, network speed and hardware restrictions. This topic is also related to the performance of the system and will be further analyzed in the performance section later. For the purpose of this test, a sampling rate of 5.5 KHz was used. Figure 7-4 shows the results from both software and hardware simulation.

```
FED_FFT: CREATING FEDERATION EXECUTION
FED_FFT: CREATION OF FEDERATION EXECUTION SUCCESSFUL
FED_FFT: ATTEMPTING TO JOIN SoundDemo
FED_FFT: SUCCESSFULLY JOINED SoundDemo
FED_FFT: Discovered object 196609
1,1
1 in last second
2,2
3,3
4,4
5,5
6,6
5 in last second
7,7
8,8
9,9
10,10
11,11
12,12
6 in last second
13,13
14,14
...
```

Figure 7-4: Message Loss Test Results

The lines that contain only two numbers (i.e. i, i) in Figure 7-4 represent the receiving and sending of the i^{th} buffer by the DSP board. According to the results shown in the figure, there is no message loss at 5.5KHz. Since the buffer size is 1024, the time taken for each buffer at 5.5 KHz sampling rate is 0.185 second, the average number of buffers received and sent is $(1/0.185)=5.6$ buffers/per second. There should be no surprise that

both simulations showed the same results, since they have the same layer 1 and layer 2 implementations that deal with RTI communication.

7.3 Performance Analysis

In this section, the performance of the spectrum analyzer is further examined. By changing the sampling rate of the soundcard, different processing loads were generated for the federations. Three aspects of performance were analyzed:

- ❑ Sampling rate limitation;
- ❑ The latency and bandwidth of the RTI;
- ❑ Hardware board and the bottleneck of the system.

7.3.1 Sampling Rate Limitation

The performance measurements of the RTI depend highly upon the characteristics of the federation. In order to generate different RTI loads for testing, the federation included the SoundSurrogate federate, DSPsurrogate federate and the display federate. By incorporating the soundcard into the simulation, different sampling rates were generated to test the performance of the RTI.

Sampling rates from 2.75 KHz to 44.1 KHz were applied in this test. The soundcard was programmed to sample at 44.1 KHz, which is the maximum possible rate for the soundcard. Lowered sampling rates were obtained by keeping a subset of the 44.1 KHz

data. For example, by keeping every 8th sample, an effective sampling rate of 5.5 KHz is achieved (i.e. 44.1 KHz ÷ 8 = 5.5 KHz). Using this technique, the sampling rate was changed at run time to sample at rates of: 2.75 KHz, 5.5 KHz, 11.02 KHz and 22.05 KHz. Because the FFT algorithm processes 1024 sample points in the case study and the original soundcard software used 2048 points sample buffers, too much rework would have been required for the case study to support analysis at a sampling rate of 44.1 KHz. To simplify the implementation, the maximum sampling rate is limited to the half of the default sampling rate which is 22.05 KHz. Table 7-1 shows the expected number of messages that should be sent out per second at the sampling rates tested.

Tested Sampling rate (KHz)	Time taken for 1024 points (s)	The expected number of 1024 point buffers sent out per second
2.75	0.372	2.7
5.5	0.185	5.4
11.02	0.092	10.8
22.05	0.046	21.7

Table 7-1: Sample Rate at Different Level

7.3.2 RTI Latency and Bandwidth

In the RTI, the internal latency is defined as the message processing time [33], which is required to completely process a sequence of the events: e.g. a federate receives RTI reflectAttributeValue or receiveInteraction call, and sends to other interested federates.

Table 7-2 shows the update rate and reflect rate recorded in a federation consisting of the SoundSurrogate, DSP and display federates. Only the soundcard hardware was included in the simulation, and the FFT algorithm was implemented in the software

federate (i.e. the stage 1 DSP federate).

Tested Sampling rate (KHz)	SoundSurrogate Fed Average Update rate per second	DSP Fed Average Reflect rate per second	Display Fed Average Reflect rate per second
2.75	2.65	2.65	2.65
5.5	5.3	5.3	5.3
11.02	10.5	10.5	10.5
22.05	21.2	21.2	21.2

Table 7-2: Object Update Rate under Different Sample Rate

Data from Table 7-2 shows that at different tested sampling rate, the average update rate and reflect rate in each federate are changing accordingly. For example, the average attribute update rate is 2.65 buffers per second at the 2.75 KHz sampling rate. 2.65 1K buffers is equal to $2.65 \times 1024 = 2.71$ KHz, which is slightly less than the expected 2.75KHz, this is due to the overhead in the PC with the soundcard that result less sampling rate produced by the soundcard. The test results closely match the expectations and each buffer containing 1024 points (2048 bytes) was delivered within 0.38 seconds ($1/2.65$) latency on a local network. After processing at the DSP federate, the buffer containing 512 magnitude floating point was then reflected at the display federate within the required time. Increasing the sampling rate also increased the update rate. When the sampling rate reached 22.05 KHz, the average update and reflection rates also increased to 21.2 buffers per second. This means that the simulation never had more than one outstanding wave update and one outstanding magnitude update waiting to be processed. The number of buffers updated per second are equal to the number of buffers that soundcard sent out. This performance data shows that the RTI's

delivery service has a sufficient bandwidth range to handle the update rates, which implies that the RTI is reliable in handling the attribute updates in this simulation.

The bandwidth in the RTI is measured as the volume of data flow during communication in the federation. Each wave update sent out from the SoundSurrogate federate is at least 2KB in size (to accommodate the sample points). Similarly, each magnitude update from the DSP federate is at least 2KB in size. When the sampling rate is 2.75 KHz, 2.65 buffers on average are sent. Therefore, when the sampling rate is 2.75 KHz, 2.65 updates are sent by each federate and the total data transmission handled every second is $(2\text{KB}+2\text{KB}) * 2.65 = 10.6$ Kbytes. With the increase of the sampling rate to 22.05 KHz, the bandwidth increased to $(2\text{KB}+2\text{KB})*21.2 = 84.8$ Kbytes per second.

7.3.3 Hardware Bottle Neck

When the DSP board is included into the stage 2 simulation, the performance of the communication between the RTI and the board is of interest. Table 7-3 shows the update rates for the HWIL simulation. This test involved the SoundSurrogate, DSPsurrogate and display federates. The DSPsurrogate federate communicated with the DSP board by sending sample points to the board and getting the magnitude values back. The FFT algorithm was processed on the DSP board.

Tested Sampling rate (KHz)	SoundSurrogate Fed Average Update rate per second	DSPsurrogate Fed Average Reflect rate per second	Display Fed Average Reflect rate per second
2.75	2.65	2.65	2.65
5.5	5.3	5.3	5.3
11.02	10.5	5.4	5.4
22.05	21.2	5.4	5.4

Table 7-3: Object Update Rate

Similar to Table 7-2, when the sampling rate is 2.75 and 5.5 KHz, the average update rate is 2.65 and 5.3 buffers per second, respectively. The reflect rate also matches the update rate. At these two sampling rates, the RTI and the DSP board communicated very well. When the sampling rate reached 11.02 KHz, the average update rate was 10.5 buffers per second, however, the reflect rate in the DSPsurrogate federate remained at 5.4 buffers per second, as did the subsequent display federate reflect rate. Figure 7-5 compares the results from both the software simulation and the HWIL simulation.

Figure 7-5: Comparison of the Performance

The reason for the performance limit at the DSPsurrogate federate is that the data from the RTI is communicated with the board using the RTDX. According to the technical data from Texas Instruments, the data transmission limit for the RTDX on the 62x DSP board is 20~50 Kbytes/second. The test data indicates that the DSPSurrogate federate performance saturated at approximately 5.4 updates per second, which corresponds to the lower end of the RTDX limits. Once saturated, arriving wave updates are queued in the network, and the latency of each subsequent update increases.

This limitation is due to the current RTDX version. There is a better version of RTDX called hi-speed RTDX [34] that can handle up to 2 Megabytes/second bandwidth. With this capacity, a sampling rate of 22.05KHz or even higher can be easily achieved.

Chapter 8 Conclusions

8.1 Conclusions

The objective of this thesis is to develop a simulation based environment that allows both software and hardware to interact early during embedded system development. Additional goals include the reuse of a generic framework for hardware subsystems, and the reuse of software code between software simulation and programmable hardware prototypes. This work has achieved the following:

- An HLA-based simulation environment is proposed and demonstrated to support the introduction of software modules and hardware prototypes.
- A two-stage development approach is defined. Stage one includes pure software simulation, where a software module simulates the behavior of the hardware. Stage two replaces the software module with the real hardware to achieve HWIL simulation.
- A four-layer interface of the HLA framework is defined and the functionality of each layer is identified. This layered design maximizes the reuse of software components and framework interface. Referring to Figure 5-1, Layers 1 and 2 remain unchanged in both development stages. The Layer 3 interface also remains unchanged, although the implementation of the interface methods must be customized in stage 2. Layer 4 requires redesign between stages.
- The design of the surrogate achieves plug and play interoperability that makes it

easy to replace software modules with hardware components.

- A case study that simulates the spectrum analyzer under the HLA framework demonstrates the proposed two-stage development method. The EVM DSP board and the soundcard were introduced into stage 2 simulations to achieve the HWIL simulation.
- A surrogate is defined to communicate with the EVM DSP board through the RTDX technology in the CCS environment. The interface is designed to be reusable among the products compatible with the CCS. Additional reusability can be achieved by reusing part of the stage 1 DSP code on the EVM DSP board.
- The correctness of the HWIL simulation is verified through different test cases and the performances of the simulation in both stages are analyzed. Stage 1 simulation can handle 22.05 KHz while stage 2 simulation could only handle 5.5 KHz. The RTDX is identified as the bottleneck of the system due to its limited transmission rate.

8.2 Future Research Directions

Suggested future work for this research includes:

- The use of an upgraded version of the RTDX channel along with the supporting hardware. Due to the limitation on the data transmission rate of the current RTDX channel, the hardware becomes the bottleneck of the system. This limited the data exchange rate between the DSP board and the RTI federation had resulted in data processing starvation. To overcome this bottleneck, the High-Speed RTDX

(HSRTDX) technology could be employed to replace the RTDX to improve the data transmission rate of the EVM DSP board. The HSRTDX provides with bandwidth of over 2 Mbytes/second and enables the application involving larger data processing and could easily allows the spectrum analyzer stage 2 simulation to process data sampled at 22KHz.

- Incorporate data intensive applications such as streaming video and hard disk drive to enhance its usage with DSP applications.
- Further development of the surrogate layer of the framework implemented in the case study to incorporate more generic hardware subsystems.
- Make use of the RTI time management system to implement a low-level clock to be synchronized with the simulation time.

Reference

- [1] Berge, J.M. "Hardware/Software Co-Design and Co-Verification". Kluwer Academic Publishers, 1997.
- [2] Altera. "Co-simulation of Embedded System Implemented in FPGAs". White paper, 2000.
- [3] Sangiovanni-Vincentelli, A. and Sifakis, J. "Embedded Software". Springer-Verlag Berlin Heidelberg New York, 2002.
- [4] Mallet, F., Boeri, F. and Duboc, J.F. "Hardware modelling and simulation using an object-oriented method". 12th European Simulation Multiconference. Manchester, United Kingdom, 1998.
- [5] Padberg, F. "A Software Process Scheduling Simulator". ICSE 2003.
- [6] Micheli, G., Ernst, R. and Wolf, W. editors, "Readings in Hardware/Software Co-Design", Morgan Kaufman Publishers, 2002.
- [7] Frakes, W.B. "Software Reuse: Advances in Software Reusability". Springer-Verlag, Computer Science Editorial III. Heidelberg, Germany, 2000.
- [8] High Level Architecture/ Run-Time Infrastructure. "RTI 1.3-Next Generation Programmer's Guide version 3.2", Department of Defense Modeling and Simulation Office, 2000.
- [9] Texas Instruments. "TMS320C6x EVM Board User's Guide". Digital Signal Processing Products, 1990.
- [10] Engelson, M. and Telewski, F. "Spectrum Analyzer Theory and Application".

Artech House Publisher, 1974.

[11] Kumar, S., Aylor, J., Johnson, B.W. and Wulf, Wm. A. "The Codesign of Embedded Systmes". Kluwer Academic Publishers, 1996.

[12] Voss, M. and e. all. "Towards a Theory for Hardware/Software Codesign". Third International Workshop on Hardware/Software Codesign, IEEE Computer Society Press. Grenoble, France, 1994.

[13] IEEE stand for modeling and simulation (M&S;) High Level Architecture (HLA) – Framework and Rules. IEEE Std. 1516-2000.

[14] High Level Architecture (HLA) Object Model Development Tool (OMDT) User's Guide. Department of Defense Modeling and Simulation Office, 1998.

[15] Texas Instruments. "Code Composer Studio Getting Started Guide". 2001.

[16] Keil, D. "Real Time Data Exchange". Digital Signal Processing Solutions, Texas Instruments, 1998.

[17] Bateman, A. "Digital Signal Processing Design". Pitman Publishing, 1988.

[18] Engelbrecht, R. "Using Hardware-in-the-loop Traffic Simulation to Evaluate Traffic Signal Controller Features". The 27th Annual Conference of the IEEE Industrial Electronics Society. 2001.

[19] Schludermann, H., Kirchmair, T. and Vorderwinkler, M. "Soft-commissioning: Hardware-in-the-loop based verification of controller software". Proceedings of the 2000 Winter Simulation Conference. 2000.

[20] Grega, W. "Hardware-in-the-loop simulation and its application in control education". 29th ASEE/IEEE Frontiers in Education Conference. San Juan, Puerto

Rico, 1999.

- [21] Lane, D.M., Falconer, G.J. and Randall, G. "Interoperability and Synchronization of Distributed Hardware-in-the-loop Simulation for Underwater Robot Development: Issues and Experiments". Proceedings of the 2001 IEEE International Conference on Robotics & Automation. Seoul, Korea, 2001.
- [22] Mello, B. and Wagner, F. "A Standard Co-simulation Backbone". VLSI-SOC, 2001.
- [23] Dubuc, F., Claude, D., Chliaev, P. and Vankov, A. "Generic Toolbox for Interoperable Systems – GTI6". EADS LAUNCH VEHICLES, 2000.
- [24] Deppe, M., Robrecht, M., Zanella, M. and Hardt, W. "Rapid Prototyping of Real-time Control Laws for Complex Mechatronic Systems". The 27th Annual Conference of the IEEE Industrial Electronics Society. 2001.
- [25] Keyhani, A., Marwali, M.N., Higuera, L.E. and Athalye, G. "An Integrated Virtual Learning System for the Development of Motor Drive Systems". IEEE Transactions on Power Systems. 2002.
- [26] Baracos, P., Murere, G., Rabbath, C.A. and Jin, W. "Enabling PC-based HIL Simulation for Automotive Applications". International Electric Machines and Drives Conference. 2001.
- [27] Wagner, F.R. et. al. "Object-Oriented Modeling and Co-simulation of Embedded Electronic Systems". VLSI: Systems on a Chip. Kluwer Academic Publishers, 2000.
- [28] Gurewich, O. and Gurewich, N. "Visual C++ 5 in 21 Days". Sams Publishing. 1997.

- [29] Texas Instruments. "RTDX help". Online technical report. URL:
<http://www.ti.com>
- [30] "AXCsound/Jcsound 3.0". Online technical report. URL:
<http://www.pipeline.com/~gogins/Csound/AXCsound.htm>
- [31] "A Scope like & Sound Input Class". Online technical report. URL:
http://www.codeguru.com/multimedia/scope_input_sound_class.shtml
- [32] "High Resolution Multifunction DAS Card Data Sheet". URL:
<http://www.modindsol.com>
- [33] Cox, A., Wood, D.D. and Petty, M.D. "Gateway Performance Using RTI 1.0".
Proceeding of the Fall 1997 Simulation Interoperability Workshop. Orlando, FL,
1997.
- [34] Fu, X. "Real-Time Digital Video Transfer via High-Speed RTDX". Texas
Instruments, Application Report, 2002.