DDS BASED MIL-STD-1553B DATA BUS INTERFACE SIMULATION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
THE MIDDLE EAST TECHNICAL UNIVERSITY

BY

ERTAN DENİZ

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

SEPTEMBER 2012

Approval of the thesis

DDS BASED MIL-STD-1553B DATA BUS INTERFACE SIMULATION

submitted by **ERTAN DENIZ** in partial fullfillment of the requirements for the degree of **Master of Science in Computer Engineering** by,

Prof. Dr. Canan Özgen
Dean, Graduate School of **Natural and Applied Sciences**

Prof. Dr. Adnan Yazıcı
Head of Department, **Computer Engineering**

Assoc. Prof. Dr. Halit Oğuztüzün
Supervisor, **Computer Engineering Dept., METU**

Dr. Umut Durak
Co-supervisor, **Informatics Institute, METU**

Examining Committee Members:

Prof. Dr. İsmail Hakkı Toroslu
Computer Engineering Dept., METU

Assoc. Prof. Dr. Halit Oğuztüzün
Computer Engineering Dept., METU

Asst. Prof. Dr. Ahmet Oğuz Akyüz
Computer Engineering Dept., METU

Dr. Onur Tolga Şehitoğlu
Computer Engineering Dept., METU

Dr. Umut Durak
Informatics Institute, METU

Date: 11.09.2012

ABSTRACT

DDS BASED MIL-STD-1553B DATA BUS INTERFACE SIMULATION

Deniz, Ertan

M.S., Department of Computer Engineering

Supervisor: Assoc. Prof. Dr. Halit Oğuztüzün

Co-Supervisor: Dr. Umut Durak

September 2012, 63 pages

This thesis describes distributed simulation of MIL-STD-1553B Serial Data Bus interface and protocol based on the Data Distribution Service (DDS) middleware standard. The data bus connects avionics system components and transports information among them in an aircraft. It is important for system designers to be able to evaluate and verify their component interfaces at the design phase. The 1553 serial data bus requires specialized hardware and wiring to operate, thus it is expensive and complex to verify component interfaces. Therefore modeling the bus on commonly available hardware and networking infrastructure is desirable for evaluation and verification of component interfaces. The DDS middleware provides publish-subscribe based communications with a number of QoS (Quality Of Service) attributes. DDS makes it easy to implement distributed systems by providing an abstraction layer over the networking interfaces of the operating systems. This thesis takes the advantage of the DDS middleware to implement a 1553 serial data bus simulation tool. In addition, the tool provides XML based interfaces and scenario definition capabilities, which enable easy and quick testing and validation of component interfaces. Verification of the tool was performed over a case study using a scenario based on the MIL-STD-1760 standard.

Keywords: MIL-STD-1553B, MIL-STD-1760, DDS, Simulation

ÖZ

DDS TABANLI MIL-STD-1553B VERİ YOLU ARAYÜZ SİMÜLASYONU

Deniz, Ertan
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. Halit Oğuztüzün
Ortak Tez Yöneticisi: Dr. Umut Durak

Eylül 2012, 63 sayfa

Bu tez Veri Dağıtım Servisi (DDS) arakatman standardı üzerinde dağıtık bir MIL-STD-1553B seri veri yolu arayüz simülasyonunu anlatmaktadır. Veri yolu uçaklardaki aviyonik sistem bileşenlerini birbirine bağlar ve aralarında veri iletişimine olanak tanır. Arayüz geliştiricileri için arayüzlerin tasarım aşamasında sınanması ve doğrulanması önemlidir. 1553 seri veri yolu özel cihazlar ve kablolama gerektirdiğinden, arayüzleri test etmek pahalı ve zor olmaktadır. Bu yüzden sistem arayüzlerini mevcut cihazlar ve ağ yapısı üzerinde test edebilmek önem kazanmaktadır. DDS arakatmanı birçok Servis Kalitesi (QoS) özellikleriyle birlikte yayımla-abone ol mimarisine uygun veri dağıtımı sunar. DDS sağladığı bu özellikleri ve işletim sisteminin ağ programlama arayüzlerini soyutlaması nedeniyle dağıtık sistemlerin geliştirimini kolaylaştırmaktadır. Bu tezde DDS arakatman standardının 1553 seri veri yolu simülasyonunun geliştirilmesinde sağladığı faydalar anlatılmaktadır. Ayrıca, geliştirilen simülasyon aracına XML tabanlı arayüz ve senaryo tanımlama kabiliyetleri eklenerek sistem arayüzlerinin kolay ve hızlı bir şekilde test edilebilmesi amaçlanmıştır. Simülasyon aracı MIL-STD-1760 standardına uygun bir senaryo ile doğrulanmıştır.

Anahtar Kelimeler: MIL-STD-1553B, MIL-STD-1760, DDS, Simülasyon

To my family...

ACKNOWLEDGMENTS

I would like to express my inmost gratitude to my supervisor Assoc. Prof. Dr. Halit Oğuztüzün for his patience, vision and understanding throughout this thesis. He was the one believing in me more than anybody else.

I am also indebted to my co-supervisor Dr. Umut Durak for his knowledge, ideas and motivation. I would be lost in this domain without his help.

I would also like to thank Kadriye Güçlü for her hard work and support in the verification process of this thesis.

I would like to express my heart-felt thanks to Hayrie, my dear wife. Without her unconditional love, joy and support, this thesis could not be completed. I also would like to thank my parents and parents in law for their complimentary love and support.

TABLE OF CONTENTS

ABSTRACT . . . iv
ÖZ . . . v
DEDICATON . . . vi
ACKNOWLEDGMENTS . . . vii
TABLE OF CONTENTS . . . viii
LIST OF FIGURES . . . xi
LIST OF TABLES . . . xiii
LIST OF ABBREVIATIONS . . . xiv
CHAPTERS
1 INTRODUCTION . . . 1
1.1 Related Work . . . 2
1.2 Motivation Behind the Proposed System . . . 4
1.3 Thesis Outline . . . 4
2 BACKGROUND . . . 6
2.1 MIL-STD-1553B . . . 6
2.1.1 History . . . 6
2.1.2 Basics . . . 7
2.1.3 Word Formats . . . 9
2.1.4 Message Formats . . . 11
2.1.5 Timing Requirements . . . 14
2.2 Data Distribution Service . . . 14
2.2.1 History . . . 14
2.2.2 Basics . . . 15

2.2.3 Conceptual Model . . . 16

2.2.4 Types, Topics and Code Generation . . . 17

2.2.5 Content Filtering . . . 18

2.2.6 Quality of Service Policies . . . 19

2.2.6.1 Reliability . . . 19

2.2.6.2 Ownership . . . 19

2.2.6.3 Destination Order . . . 19

2.2.6.4 Durability . . . 20

2.2.6.5 History . . . 21

2.2.6.6 Lifespan . . . 21

2.2.6.7 Deadline . . . 21

2.2.6.8 Resource Limits . . . 21

2.2.6.9 Time Based Filter . . . 22

3 DDS BASED MIL-STD-1553B SIMULATION IMPLEMENTATION . . . 23

3.1 Development Environment . . . 23

3.2 Design . . . 23

3.2.1 BusFactory (public) . . . 24

3.2.1.1 Instance . . . 24

3.2.1.2 CreateBC . . . 24

3.2.1.3 CreateRT . . . 24

3.2.1.4 RegisterBCImplementation . . . 25

3.2.1.5 RegisterRTImplementation . . . 25

3.2.2 IBusController (public) . . . 25

3.2.2.1 CreateMessage . . . 25

3.2.2.2 AlterMessage . . . 25

3.2.2.3 CreateFrame . . . 25

3.2.2.4 StartFrame . . . 26

3.2.2.5 Run 26
3.2.3 IRemoteTerminal (public) 26
3.2.3.1 AssignRxBlock 26
3.2.3.2 AssignTxBlock 26
3.2.3.3 Run 26
3.2.4 SimBusController (private) 26
3.2.5 SimRemoteTerminal (private) 27
3.3 DDS Data Types 27
3.4 Advantages of Using DDS in the Simulation Tool 28
3.4.1 Publish-Subscribe and Automatic Discovery 28
3.4.2 Content Filtering 29
3.4.3 Command/Response with WaitSet 29
3.5 Deployment 30
3.6 XML Scenario Interface 30
4 CASE STUDY: A BASIC MIL-STD-1760 SIMULATION 34
5 PERFORMANCE ANALYSIS 41
6 CONCLUSIONS 45
REFERENCES 47
APPENDICES
A EXAMPLE BUS CONTROLLER XML FILE 50
B EXAMPLE REMOTE TERMINAL XML FILE 52
C XML SCHEMA FOR BUS CONTROLLER 53
D XML SCHEMA FOR REMOTE TERMINALS 57
E CASE STUDY BUS CONTROLLER XML FILE 59
F CASE STUDY REMOTE TERMINAL 1 XML FILE 62
G CASE STUDY REMOTE TERMINAL 2 XML FILE 63

LIST OF FIGURES

FIGURES

Figure 2.1 Legacy Point-to-Point Architecture of Avionics Systems [1]. 6
Figure 2.2 Data Bus Architecture [1]. 7
Figure 2.3 Structure of the MIL-STD-1553B Serial Data Bus [2]. 9
Figure 2.4 MIL-STD-1553B Word Formats [3]. 10
Figure 2.5 MIL-STD-1553B Data Encoding [1]. 10
Figure 2.6 MIL-STD-1553B Information Transfer Formats [4]. 12
Figure 2.7 MIL-STD-1553B Broadcast Information Transfer Formats [4]. 13
Figure 2.8 DDS Conceptual Model [5]. 16
Figure 2.9 DDS Entities [6]. 17
Figure 2.10 Generated classes for the "Track" data type [7]. 18
Figure 2.11 DDS QoS policies [5]. 20

Figure 3.1 Class Diagram of the 1553 data bus simulation library. 24
Figure 3.2 Example Deployment of a bus controller and a single remote terminal on different computers. 30
Figure 3.3 The XML Schema for the Bus Controller XML Files. 32
Figure 3.4 The XML Schema for the Remote Terminal XML Files. 33

Figure 4.1 Store Control message defined in MIL-STD-1760. 36
Figure 4.2 Store Monitor message defined in MIL-STD-1760. 37
Figure 4.3 Store Description message defined in MIL-STD-1760. 38
Figure 4.4 Aircraft Description message defined in MIL-STD-1760. 39
Figure 4.5 Sequence diagram of the MIL-STD-1760 scenario. 40

Figure 5.1 Performance Test of a BC2RT message on the 1553 data bus simulation tool. 42

Figure 5.2 Performance Test of a RT2BC message on the 1553 data bus simulation tool. 42
Figure 5.3 Performance Test of a RT2RT message on the 1553 data bus simulation tool. 43

Figure 6.1 Integration of legacy 1553 components into an AFDX based network. . 46

LIST OF TABLES

TABLES

Table 2.1 Summary of MIL-STD-1553B Characteristics [1] 8

Table 5.1 Performance Test Setup 41

Table 5.2 Performance Test Results 43

Table 5.3 Message Latencies on 1553 data bus 44

LIST OF ABBREVIATIONS

AEIS	Aircraft/Store Electrical Interconnection System
AFDX	Avionics Full-Duplex Switched Ethernet
API	Application Programming Interface
ATM	Asynchronous Transfer Mode
CORBA	Common Object Request Broker Architecture
DDS	Data Distribution Service
DCPS	Data Centric Publish Subscribe
DLRL	Data Local Reconstruction Layer
IDL	Interface Description Language
IEEE	Institute of Electrical and Electronics Engineers
MSB	Most Significant Bit
OACE	Open Architecture Computing Environment
OMG	Object Management Group
OSCI	Open SystemC Initiative
RTPS	Real-Time Publish Subscribe
QoS	Quality of Service
SQL	Structured Query Language
XML	Extensible Markup Language
AS	Action Scheduling
BC	Bus Controller
RT	Remote Terminal
BM	Bus Monitor

CHAPTER 1

INTRODUCTION

MIL-STD-1553B, referred as 1553 hereafter, is a military standard published by the United States Department of Defense that establishes requirements for digital, command/response, time division multiplexing (Data Bus) techniques [3]. First published in 1970s, the standard describes mechanical, electrical, and functional characteristics of a serial data bus.

The 1553 data bus was successfully applied to many military avionics and spacecraft subsystems since its introduction and became an industrial standard. The standard requires specialized hardware and wiring to be able to evaluate and verify subsystem interfaces. Nonetheless, use of real 1553 hardware within simulation and development environments can be both very costly and unnecessarily restrictive [8].

During interface definition phases, it's essential to concentrate on the interfaces between components, not on the hardware and wiring details. This thesis describes a MIL-STD-1553B interface and protocol simulation, which provides an easy way to verify the designed component interfaces. The simulation tool provides a way to define interfaces as XML files compliant to the MIL-STD-1553B meta-model. Moreover, a scenario meta-model is defined so that test scenarios can be written and verified through the simulation tool.

This thesis aims to implement a maintainable and extendable 1553 data bus simulation on commonly available hardware and networking infrastructures. The purpose is to make the simulation tool easily accessible on existing operating systems and networking infrastructures at reduced costs. For this purpose, the DCPS (Data Centric Publish Subscribe) layer of DDS (Data Distribution Service) is employed as the transport mechanism to implement the 1553 data bus simulation. DDS is a middleware standard published by OMG (Object Management Group) [5]. It provides real time publish-subscribe communications to distributed applications over Ethernet networks. It has a number of QoS (Quality Of Service) settings that enable configuring the middleware according to the specific needs of each component in a distributed system. There are several implementations of DDS, including two mature open source products.

The verification of the simulation tool is done by a case study based on a MIL-STD-1760 scenario. MIL-STD-1760 is a standard defining electrical interfaces between a military aircraft and its stores. MIL-STD-1760 based scenarios give a realistic approach and provide a good testing platform for the 1553 data bus simulation. The scenarios are defined in XML files which are generated by another tool specifically developed to design MIL-STD-1760 scenarios [9].

1.1 Related Work

MIL-STD-1553B data bus originated from the need of an infrastructure to develop systems that have components distributed over the various parts of the aircrafts. Initially direct point-to-point wires were used to connect components. But when the complexity of the systems deployed in the aircrafts began to increase, the wiring became both very complex and heavy. To overcome these difficulties, The US Department of Defense published the 1553 serial data bus standard and chose multiplexing because of the following advantages [10]:

- Weight reduction
- Simplicity of system design
- Standardisation
- Flexibility

The introduction of the standard enabled many new sensors and subsystems to be integrated into the aircrafts, which led to interfaces between components to be more and more complex. The 1553 data bus requires special hardware and wiring which makes it both costly and restrictive to develop component interfaces. Traditionally the approach to these kinds of problems is to develop simulation systems so that interface designers can verify their designs without the need of expensive hardware or complex cabling. It is not different in this case and many simulation approaches have been proposed for the MIL-STD-1553B data bus [8] [11] [2] [12].

N. Downing [8] proposed a virtual 1553 data bus, of which the idea is the move functionality from hardware to software. The 1553 cards are replaced by software simulation and as the transport mechanism commonly available networking technology is used. TCP/IP networks are standard and Ethernet network interface is available in every computer. Virtual 1553 data bus essentially is a software library that provides a programming interface compatible with the MIL-STD-1553B standard. The library simulates the behavior of a real 1553 card and transmits messages over TCP/IP networks. MIL-STD-1553B defines very strict timing requirements which are chosen not to be implemented in this virtual 1553 data bus. The approach is chosen so that the simulation library is portable and can be run on any operating system, not necessarily real-time. This thesis takes this idea and improves it by introducing the DDS middleware as the transport mechanism.

Another approach by J. Tian et al [11] is to simulate the transport layer protocol of MIL-STD-1553B on a single computer with shared-memory as the transport mechanism between bus controller and remote terminals. The problem is approached as a discrete event system and is simulated with Action Scheduling (AS). Action Scheduling is a simulation method in which actions describe the result as the system state changes. Therefore, dynamic behavior is initiated by actions in this approach. This thesis describes a distributed simulation of MIL-STD-1553B terminals which can provide a better representation of a real 1553 environment.

SystemC is a modeling platform consisting of C++ classes and macros that include an event-driven simulation kernel. It is developed by OSCI (Open SystemC Initiative) and has been approved as an IEEE standard, namely IEEE 1666-2011 [13]. S.M. Aziz [2] described a transaction level SystemC model of the 1553 data bus that provides cycle-accurate simulation. The model uses a clock-based synchronization strategy to achieve cycle-accurate performance estimates.

Engblom and Holm [12] discuss a fully virtual multi-node 1553 bus computer system simulated using the Virtutech Simics [14] simulator framework. The aim is to provide an alternative development and prototyping environment to software developers so that unmodified software can be run on the simulation. The simulation environment contains multiple nodes which are connected using a simulated 1553 bus. The 1553 data bus is simulated on the message level.

Another interesting research is by D. Parish et al [15], which aimed at resolving the problem of higher communication requirements of future aircraft avionics systems. The paper describes an incremental approach to replace MIL-STD-1553B data bus with a switched network technology from the Telecommunications area, Asynchronous Transfer Mode (ATM). The major step required to replace the data bus of the current avionics systems is to emulate the 1553 data bus on an ATM network. It is shown that the emulation of 1553 data bus over the ATM network allow future ATM compliant equipment to coexist within the same network as 1553 elements. This thesis focuses on widely available Ethernet networks instead of ATM networks. Moreover, unlike the idea of emulating the 1553 data bus on a real production system, this thesis aims to develop a simulation tool to be used in development environments. However, the idea of emulating 1553 on production environments may be researched as a future work of this thesis.

1.2 Motivation Behind the Proposed System

Designing and testing components that interface with the MIL-STD-1553B data bus requires expensive hardware and special cabling, which is both costly and complex at the same time. Interface designers face the issue that they have to finalize the cabling before starting to implement and test any of the interfaces with MIL-STD-1553B. It was this reason that Downing proposed a virtual MIL-STD-1553B bus that will be implemented on software and provide the bus protocol without the need for expensive cards and special cabling between components [8]. This virtual 1553 bus is essentially a simulation of the programming interface and the protocol of the bus.

Ethernet is a widely deployed local area networking technology. It's available on nearly every computer and setting up a local area network with Ethernet is very cheap and easy. While choosing Ethernet as the transport protocol was not a hard choice at all, still there were some questions regarding how to implement the simulation over TCP/IP. Every operating system has its own programming interfaces to write applications that connect to each other via TCP/IP. This makes it hard for implementing a portable simulation tool.

DDS middleware provides data-centric and publish-subscribe based data distribution with easy to use API that is portable across operating systems and hardware architectures. Moreover, a wide selection of QoS settings, content filtering and automatic discovery of participants makes it easy to develop distributed systems and provides great flexibility in the deployment. Using an already proven technology to implement the simulation tool also contributes to the extendability of the tool and lowering the maintenance efforts in the future.

1.3 Thesis Outline

The rest of this thesis is organized as follows. In Chapter 2, the background information on which this thesis is based on is provided. MIL-STD-1553B data bus and the DDS middleware standard are discussed in detail. The design of the simulation tool implementation is discussed in Chapter 3. It contains information about how DDS is used and details the design of the XML scenario definition model. Chapter 4 includes a case study based on a MIL-STD-1760 simulation and Chapter 5 contains performance tests of the simulation tool. Finally, Chapter 6 discusses the accomplishments and limitations of the work performed in this thesis and points out some future directions.

CHAPTER 2

BACKGROUND

2.1 MIL-STD-1553B

2.1.1 History

Military aircraft before 1970s were developed by integrating various avionics subsystems by direct point-to-point wires. Inserting new subsystems or removing existing ones posed great problems because of this direct wiring. As more and more subsystems were added the aircrafts became more complex and the overall weight increased [1]. The need for a networking standard emerged from these problems. In 1968, a subcommittee established by SAE (Society of Automotive Engineers) started working on a standard, which aimed to define a serial data bus to meet the requirements of military avionics systems.

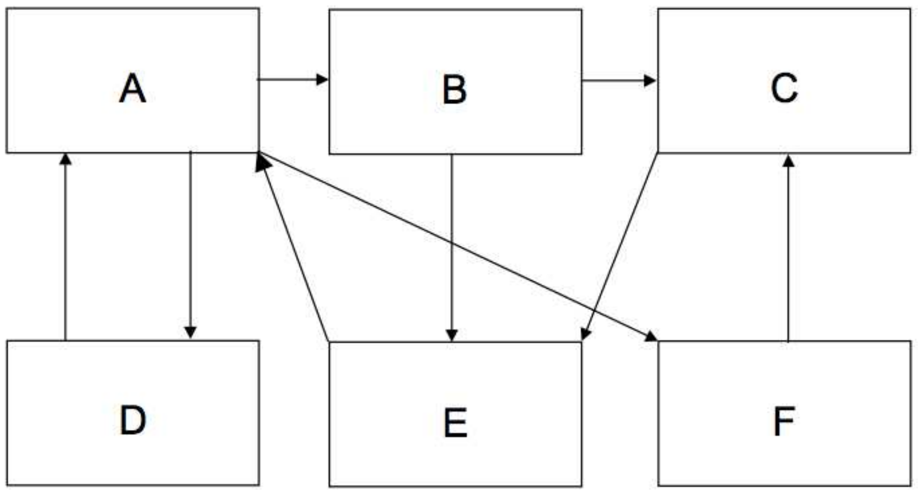


Figure 2.1: Legacy Point-to-Point Architecture of Avionics Systems [1].

MIL-STD-1553 is a military standard that defines mechanical, electrical and functional characteristics of a serial data bus. The initial version of MIL-STD-1553 was released in August of 1973. The first system using the standard was F16 - Fighting Falcon fighter aircraft.

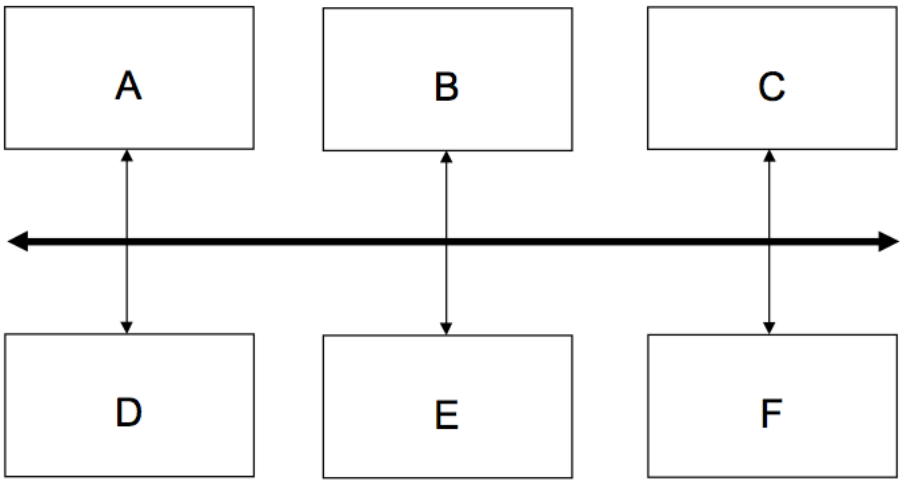


Figure 2.2: Data Bus Architecture [1].

After further improvements a revision named MIL-STD-1553A was released in 1975 and it was used both in F-16 and the AH-64A Apache attack helicopter. Those real world experiences triggered more changes in the standard, which were primarily to establish a better standard by specifying the electrical interfaces explicitly rather than leaving them to the users. The improvements assured electrical compatibility between designs by different manufacturers. This new version MIL-STD-1553B was released in 1978.

The standard received some further changes as notices, but the version still remained at the same “B” level. While it was originally designed for military avionics, the standard also found wide usage in both military and civil systems. It has been deployed in satellites, missiles, bombers, tanks, ships, space shuttles, power systems, control systems, passenger cars, oil platforms, etc. Notice 2 which was published in 1986 removed all references to aircraft and airborne, which recognized and justified the adoption of the standard in other areas than avionics.

2.1.2 Basics

The MIL-STD-1553B bus features a time division multiplexing, half duplex command/response protocol which runs on a single twisted shielded pair of wires. Only a single computer is allowed to transmit on the bus at a given time. On the other hand, full duplex systems like Ethernet and RS-232/422 have multiple cables to provide simultaneous transmit and receive. The data rate on a 1553 bus is 1 Mhz. Fault tolerance is supported by dual redundancy. A summary of MIL-STD-1553B characteristics can be found in Table 2.1.

Table 2.1: Summary of MIL-STD-1553B Characteristics [1]

Data Rate	1 MHz
Word Length	20 bits
Data Bits / Word	16 bits
Message Length	Maximum of 32 data words
Transmission Technique	Half-duplex
Operation	Asynchronous
Encoding	Manchester II bi-phase
Protocol	Command/response
Bus Control	Single or Multiple
Fault Tolerance	Typically Dual Redundant, second bus in "Hot Backup" status
Message Formats	Controller to terminal Terminal to controller Terminal to terminal Broadcast System control
Number of Remote Terminals	Maximum of 31
Terminal Types	Remote terminal Bus controller Bus monitor
Transmission Media	Twisted shielded pair
Coupling	Transformer and direct

A 1553 bus system architecture consists of multiple computers having a master/slave relationship. The computer that acts as the master and controls all the communication on the bus is called Bus Controller (BC). The BC can control multiple slave computers which are called Remote Terminals (RT) by sending commands to them. There may also be one or more passive Bus Monitors (BM) deployed on the bus which are only used to monitor or record the messages on the bus but can't transmit any messages [16].

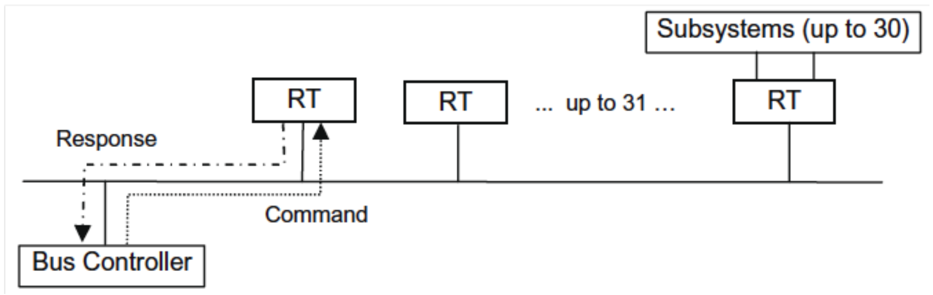


Figure 2.3: Structure of the MIL-STD-1553B Serial Data Bus [2].

There can be thirty-one remote terminals connected to the bus in addition to the bus controller. Remote terminals receive commands from the bus controller and respond according those the commands. Only the bus controller can initiate a transmission on the bus.

2.1.3 Word Formats

There are three different words that form the messages that are transmitted on the bus; command word, data word and status word. Each word is formed by a three-bit time sync, sixteen bits for the information field itself, and a parity bit at the end, which makes a total of twenty bits. Each bit is timed as one microsecond, resulting in one megabit per second transmission rate for the bus [8]. Figure 2.4 shows an illustration of the three word formats.

The sixteen bit information fields of words are encoded using a bi-phase Manchester II format. The Manchester II encoding is shown on Figure 2.5. A logic "1" is represented by a 0.5 μs high to a 0.5 μs low transition, and a logic "0" is represented by the opposite low to high transition. As shown on Figure 2.4 each word is preceded by a three-bit sync which is encoded as 1.5 μs low and 1.5 μs high for data words and opposite 1.5 μs high and 1.5 μs low for command and status words.

The command word can be transmitted only by the bus controller and as its name implies it contains a command to the remote terminals to perform. The sixteen bits of command payload contains five bits for the remote terminal address field, single bit for the command type field, five bits for the subaddress/mode field and five bits for the word count/mode

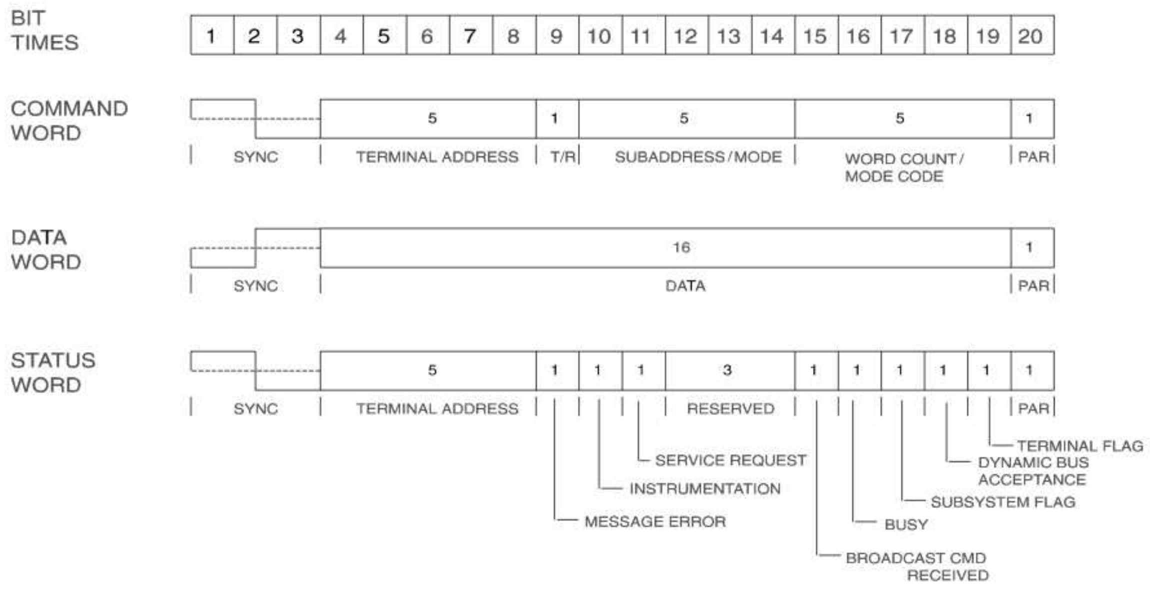


Figure 2.4: MIL-STD-1553B Word Formats [3].

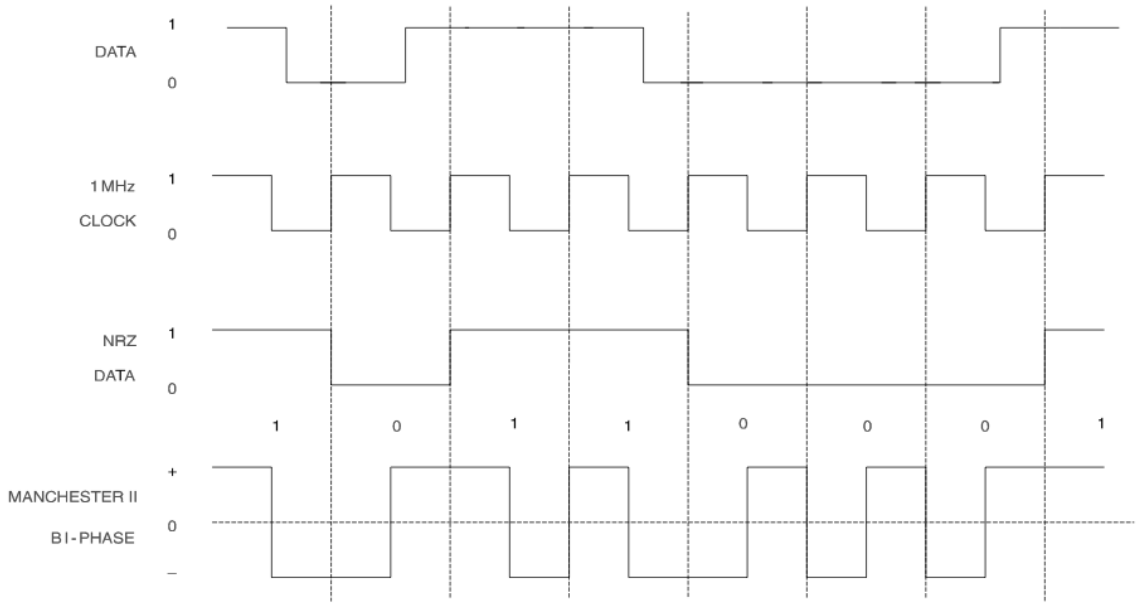


Figure 2.5: MIL-STD-1553B Data Encoding [1].

code field. Each remote terminal has a unique address so the first five bits can uniquely adress them. The address of 31 (11111) is reserved as the broadcast address, so a maximum of thirty-one remote terminals are supported. The one-bit command type represents the action that the remote terminal should perform; which is either a logic "1" for transmit or a logic "0" for receive. The five bit subaddress is used to direct the command to different functions within the subsystem [1]. Binary values 00000 and 11111 are reserved and they indicate that the command is a Mode Code. The last five bits represent the number of words to be transmitted or received. Binary value 00000 is interpreted as thirty-two words so a maximum of thirty-two words can be transmitted and received with a single message in a 1553 data bus. In case of a Mode Code command, the last five bits represent the mode code to be performed.

The data word contains the actual information that is transferred within a message [1]. It is transmitted by the bus controller after it sends a receive command or by the remote terminals after they receive a transmit command. The sixteen bits of payload of the data word is application specific and is defined by the interface designers. The standard only requires that the most significant bit (MSB) of the data must be transmitted first [1].

The status word is used to indicate the status of the remote terminal to the bus controller. Remote terminals must send a status word as the first word of a response to a valid message from bus controller. The only time when a status word is suppressed is when the optional broadcast operation is performed [3]. The sixteen bits of status word payload contains five bits remote terminal address, message error bit, instrumentation bit, service request bit, three reserved bits, broadcast command receive bit, busy bit, subsystem flag bit, dynamic bus control acceptance bit and terminal flag bit [3]. The remote terminal address field contains the unique address of the remote terminal sending the status word. Message error bit indicates whether some data words that are received failed remote terminal's validity tests. Service request bit is used to indicate exceptional data transmission needs from the remote terminals. It is used when the remote terminal or one of it's subsystems wants to transmit data which is not periodic. This is because the remote terminals can't start a transmission by their own, so they set the service request bit and the bus controller then sends a transmit command if needed.

2.1.4 Message Formats

The MIL-STD-1553B standard strictly defines six information transfer formats and four broadcast information transfer formats. No other message formats are allowed to be used on the data bus. Figure 2.6 shows the six message formats and Figure 2.7 shows the remaining four broadcast message formats defined by the standard.

The six message formats allowed between the bus controller and a remote terminal (or a pair of terminals) can be summarised as the following (see Figure 2.6):

1. **Bus controller to remote terminal** transfers are triggered by a receive command which is immediately followed by some data words without any gaps in between. The

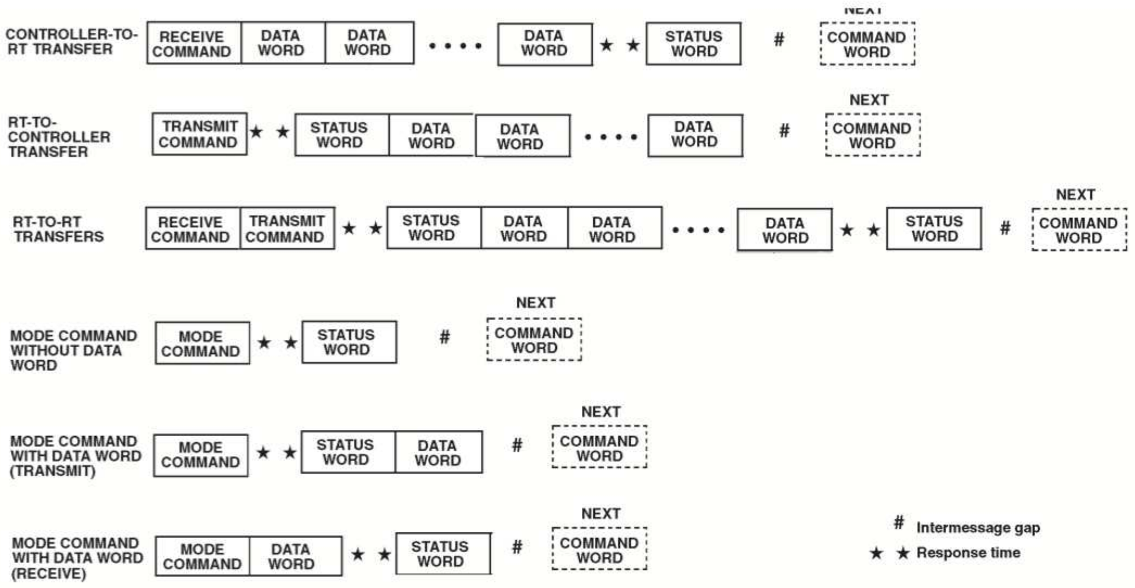


Figure 2.6: MIL-STD-1553B Information Transfer Formats [4].

number of data words is specified in the command word. The remote terminal then transmits a status word back to the controller.

2. **Remote terminal to bus controller** transfers are triggered by a transmit command from the bus controller. The remote terminal then responds with a status word which is immediately followed by some data words without any gaps between them.

3. **Remote terminal to remote terminal** transfers are triggered by a receive and transmit commands which are send from the bus controller without any gaps in between. The receive command is addressed to the remote terminal which will listen for the data, and the transmit command is addressed to the remote terminal that will send the data. The remote terminal that receives the transmit command sends a status word which is immediately followed by some data words without any gaps between them. Finally, the listening remote terminal which received the data sends a status word.

4. **Mode command without data word** transfers are triggered by a mode command and finalised by a status word from the remote terminal.

5. **Mode command with data word (transmit)** transfers are triggered by a transmit command that contains a mode code. The remote terminal responds with a status word and a single data word without any gaps in between.

6. **Mode command with data word (receive)** transfers are triggered by a receive command that contains a mode code immediately followed by a single data word without any gaps in between. The remote terminal responds with a status word.

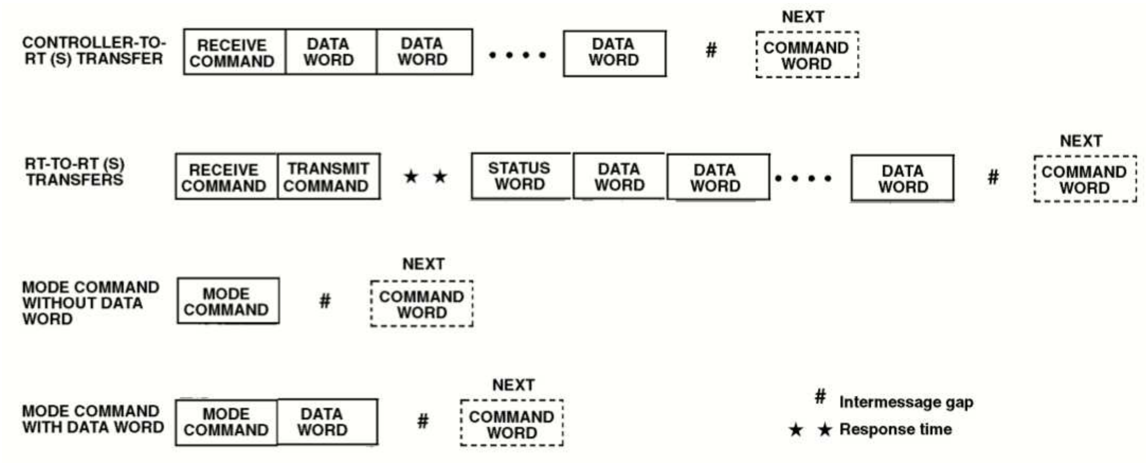


Figure 2.7: MIL-STD-1553B Broadcast Information Transfer Formats [4].

The four broadcast message formats allowed between the bus controller and remote terminals can be summarised as the following (see Figure 2.7):

1. **Bus controller to remote terminal(s) (broadcast)** transfers are triggered by a receive command addressed to 31 (11111) immediately followed by some data words without any gaps in between. Remote terminals do not send back a status word as this may cause conflicts on the data bus.

2. **Remote terminal to remote terminal(s) (broadcast)** transfers are triggered by a receive command addressed to 31 (11111) immediately followed by a transmit command addressed to a specific remote terminal. The remote terminal that received the transmit command sends a status word immediately followed by data words with no gaps in between. The other remote terminals that received the data does not send a status word.

3. **Mode command without data word (broadcast)** transfers are triggered by a transmit command addressed to 31 (11111) that contains a mode code. The receiving remote terminals does not send a status word.

4. **Mode command with data word (broadcast)** transfers are triggered by a receive command addressed to 31 (11111) that contains a mode code immediately followed by a single data word without no gaps in between. The remote terminals does not send a status word.

2.1.5 Timing Requirements

Avionics systems that are targeted by the MIL-STD-1553B require predictable and deterministic data transmission between its subsystems. Therefore, the standard defines some specific timing requirements to be able to provide predictable data transfers for the real-time systems it's designed for. The timing requirements specified in the standard can be summarized as the following:

- Inter message gap time: According to the standard, the bus controller shall provide a minimum gap time of 4.0 μs between messages [3]. This inter message gap time is usually higher than the minimum specified in the standard.
- Response time: The remote terminals shall respond to valid command word within time period of 4.0 to 12.0 μs [3].
- Minimum no-response timeout: This timeout is defined to be 14.0 μs. It indicates the time to wait before considering that a response has not occurred either because the remote terminal did not receive the previous command or it is unable to respond.

2.2 Data Distribution Service

2.2.1 History

Data Distribution Service for Real-Time Systems (DDS) standard is a relatively new standard, which was established by the Object Management Group (OMG) in 2003. The standard was a joint effort by two major proprietary DDS vendors, Real-Time Innovations from America and Thales Group from France. Thales Group later handed over their OpenSplice DDS product development to PrismTech.

Shortly after the standard was published, The US Navy has released their Open Architecture Computing Environment (OACE) [17] specification in 2004. OACE design documents mandated the use of CORBA and DDS standards for the implementation of ship command and control systems. This greatly increased the acceptance of the standard in the defense software industry. This led to other companies implementing their own DDS products. The first DDS compliant middleware implemented by third party other than the publishers of the standard was a company called MilSOFT from Turkey [18].

The standard later received some minor modifications, mostly corrections of typing errors. The most current version is 1.2 which was published in 2007. When the standard was first released it defined only the behavior of the middleware and the standard API it provides to the applications. OMG continued working on improving the interoperability and extending the standard. Those extensions were published as separate specifications. A major milestone was achieved in 2006, when the Real-time Publish-Subscribe Wire Protocol Specification (RTPS) [19] was published. RTPS defines the low level wire protocol which enables different DDS products from multiple vendors to be able to interoperate on the same network.

2.2.2 Basics

DDS standard defines a middleware with a data centric and publish-subscribe architecture for real-time systems [5]. The data centricity of the middleware distinguishes it from message centric or service based middleware standards. It is responsible to transport data from publishers to subscribers and maintain a coherent state of data among nodes. The publish-subscribe architecture enables flexibility and scalability in distributed systems development. In such systems, data producers and consumers are decoupled, which enables easy integration of new modules without the need to reconstruct the rest of the system.

Real-time distributed systems expect predictable distribution of data with minimal overhead. Since the resources are limited and each data may have different resource requirements, it's essential for a middleware to have configuration parameters [5]. DDS defines these configuration parameters as QoS (Quality of Service). The wide range of QoS parameters allows DDS middleware to be configured according to the specific requirements of each component in the system.

The DDS specification describes two layers of programming interfaces:

- Data Centric Publish-Subscribe (DCPS) layer is the core of DDS and is targeted towards the efficient delivery of the proper information to the proper recipients.

- Data Local Reconstruction Layer (DLRL) is an optional layer that may be built on top of DCPS and allows for a simple integration of DDS into the application layer by reconstructing data items into object oriented interfaces.

In this thesis, only the DCPS layer of DDS is utilized and the following sections which contain information about the standard, refer to the DCPS layer of DDS.

2.2.3 Conceptual Model

DDS standard provides an API (Application Programming Interface) which is object oriented. The most important objects are DomainParticipant, Topic, Publisher, Subscriber, DataWriter and DataReader. See Figure 2.8 for a representation of the DDS conceptual model.

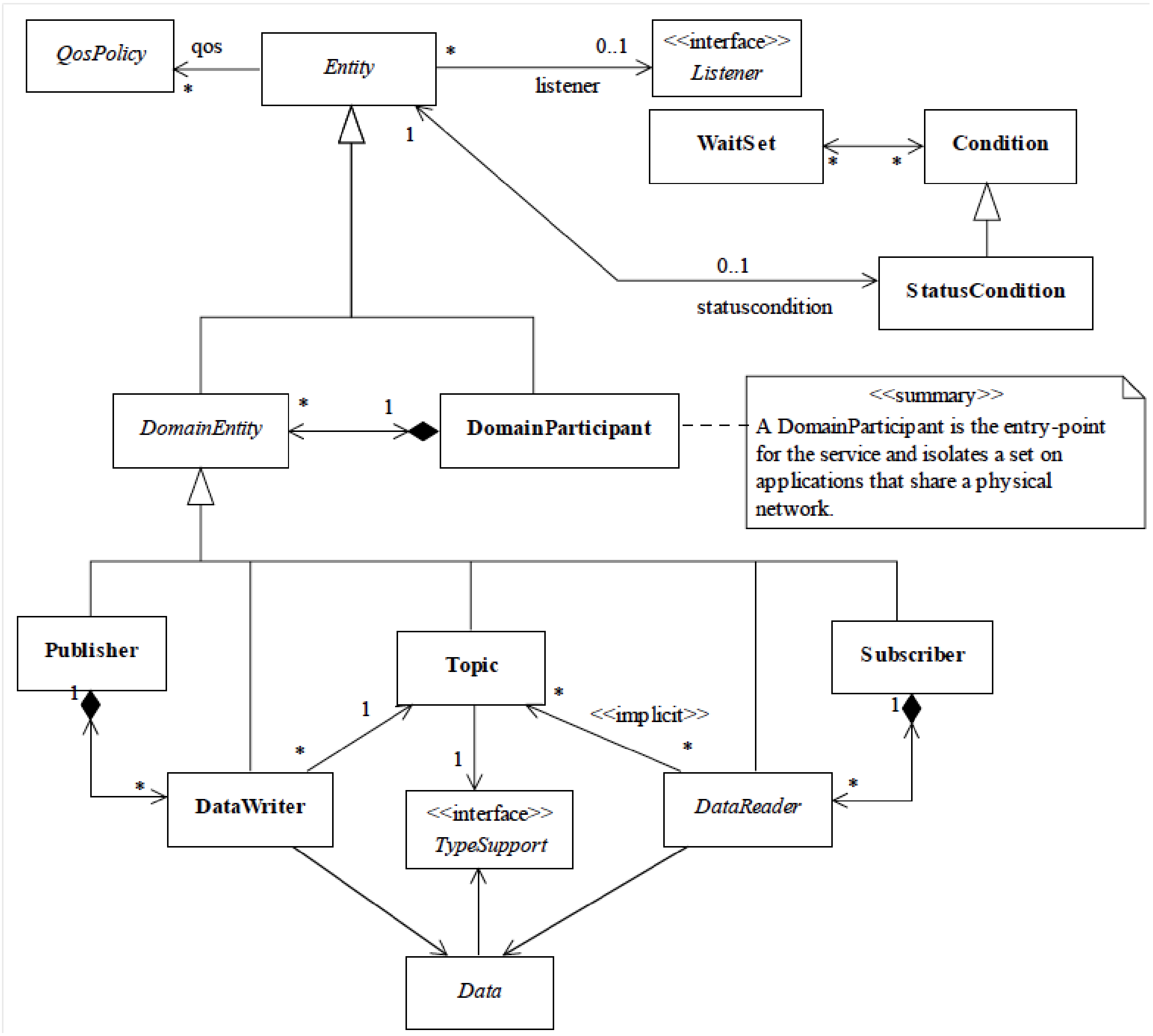


Figure 2.8: DDS Conceptual Model [5].

DDS defines a Domain as a virtual network of applications communicating with each other. Only applications which are members of the same Domain can communicate with each other. The Domains are specified by an integer and applications participate to the desired domains by creating a DomainParticipant object. The data types that are published are defined by TypeSupport objects. Topic entities are basically named data types with specific QoS properties attached. Each Topic can have only one associated data type, but multiple Topics can have the same data type, in which case the Topics are distinguished by their name and QoS parameters. Topics are the entities that are being published and subscribed to by the applications.

Applications create Publisher objects through the DomainParticipant interface to be able to publish data to the system. Publisher objects group and manage a set of DataWriters. DataWriters are created through the interface of Publisher objects. Each DataWriter is associated with only one Topic object and it can publish data only for that Topic. Likewise, applications create Subscriber objects using the interface of DomainParticipant. Subscriber objects group and manage a set o DataReaders. DataReaders are created through the interface of Subscriber. Each DataReader is associated with only one Topic object and it can receive data only for that Topic. See Figure 2.9 for an illustration of DDS Entities.

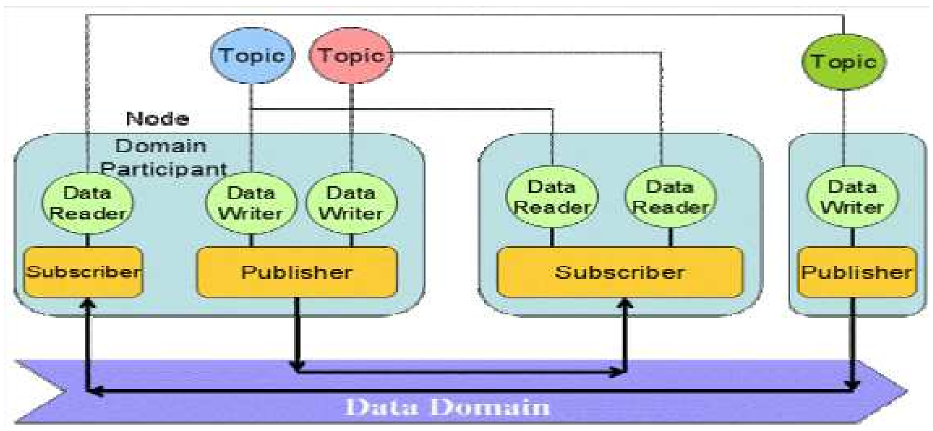


Figure 2.9: DDS Entities [6].

2.2.4 Types, Topics and Code Generation

Data types defined by DDS users may contain special members that are used to distinguish between successive updates of the data. Those special members are called the "Key" of the data. Different data values that have the same Key value are called "Instance". Successive updates of a data that has the same Key value are members of the same Instance and each of those updates are called "Sample". If no key is provided for the data type, then the data set associated with the Topic is restricted to a single Instance [5].

DDS standard defines a code generation stage so that type specific interfaces are generated for interfacing with the middleware. Software developers are supposed to define their data types with a subset of the OMG IDL (Interface Definition Language) [20]. The code generators then generate type specific TypeSupport, DataWriter and DataReader classes derived from their base classes. Figure 2.10 shows generated classes for the "Track" data type. The Track data type that's defined in IDL has its trackId field set as the Key of the data.

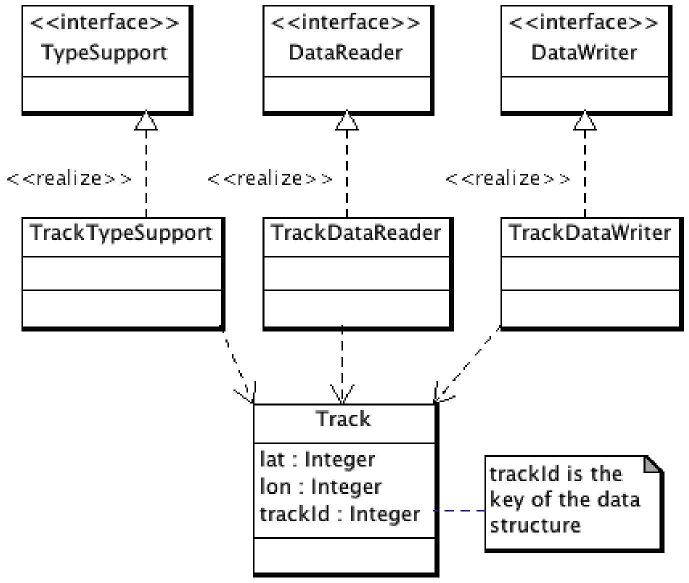


Figure 2.10: Generated classes for the "Track" data type [7].

2.2.5 Content Filtering

DDS middleware is fully aware of the data content that's being transmitted between applications. This is a characteristic feature of data centric architecture that distinguishes it from message centric or service based architectures. Applications can specify only the data they are willing to receive and the middleware filters unnecessary data. This content based filtering is achieved by a subset of SQL [5]. A DataReader associated with a ContentFilteredTopic receives only the desired data specified in the ContentFilteredTopic creation. On the other hand, a DataReader associated with a normal Topic will receive all data published to that Topic, but applications can later query the DataReader cache with SQL.

2.2.6 Quality of Service Policies

Being able to specify different QoS policies for each individual Topic, DataReader and DataWriter is the essence of data centricity within DDS [6]. DDS provides twenty-two different QoS policies. Distributed systems contain many different components whose communication requirements differ from each other. The combination of QoS parameters enable system architects to adapt the middleware to the desired needs of each component in the distributed system. Figure 2.11 shows all of the supported QoS policies and their parameters by DDS.

2.2.6.1 Reliability

RELIABILITY QoS policy indicates the level of reliability requested by a DataReader or offered by a DataWriter [5]. This policy can take two values; BEST_EFFORT or RELIABLE. BEST_EFFORT indicates that the middleware should do its best to transmit the data to the DataReader, but it's not required to guarantee the delivery. RELIABLE indicates that the middleware will guarantee the delivery to the DataReader, meaning that the data will be retransmitted if it's lost on the network.

2.2.6.2 Ownership

OWNERSHIP controls whether multiple DataWriter entities can update the same instance of data. When SHARED ownership kind is used multiple DataWriters are enabled to update a data instance. EXCLUSIVE ownership kind allows only the strongest DataWriter to update data instances. The strength of DataWriters is specified by the OWNERSHIP_STRENGTH parameter.

2.2.6.3 Destination Order

DESTINATION_ORDER can be used to specify an ordering at the receiving side. Since in a distributed environment, data instances published by multiple publishers may arrive in a different order then they are published this QoS policy enables the choice of ordering either by the source timestamp or by the receiving timestamp.

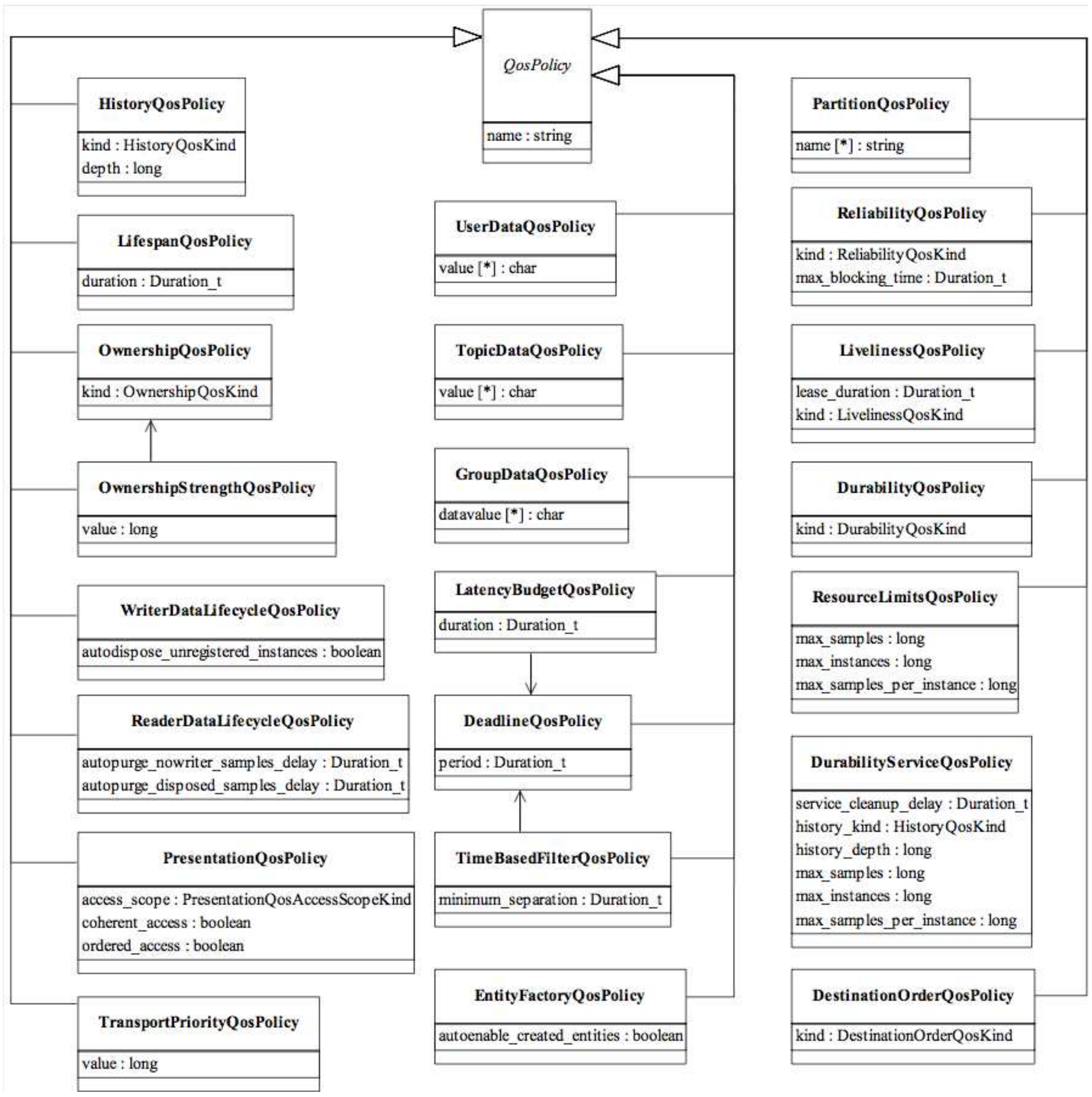


Figure 2.11: DDS QoS policies [5].

2.2.6.4 Durability

DURABILITY QoS policy provides the ability of data to be decoupled from the time it was published, by making it available to DataReaders that join the network after the data was published. The policy can take four values; VOLATILE, TRANSIENT_LOCAL, TRANSIENT and PERSISTENT. VOLATILE indicates that once the data is published, late joining DataReaders will not be able to receive the data, regardless of the availability of the DataReader at that time. TRANSIENT_LOCAL enables previously published data to be available to late joining DataReaders, but that data is bound to the lifecycle of DataWriter. When the DataWriter is destroyed TRANSIENT_LOCAL data is also no longer available.

TRANSIENT data is available to late joining DataReader even if the DataWriter is destroyed. The data is simply stored in the memory by the middleware services. TRANSIENT data disappears when the systems are shut down. PERSISTENT data is stored on disk and is available even after a system restart.

2.2.6.5 History

HISTORY specifies how many data samples will be stored for later delivery by the DDS infrastructure [6]. The applications can decide to keep the last N samples of each instance by using the KEEP_LAST history kind. KEEP_ALL history kind can be used to keep all of the previously published samples for each instance. This QoS policy can be used as a buffering mechanism for the RELIABILITY QoS. It can also be used with the combination of the DURABILITY QoS to configure the amount of historical data stored in the middleware for providing it to late joining DataReaders.

2.2.6.6 Lifespan

LIFESPAN QoS policy enables to set a time for validity of data samples. When the specified time expires the data samples are discarded and they can not be accessed any more.

2.2.6.7 Deadline

DEADLINE can be used in cases where a Topic is expected to have each instance updated periodically [5]. DataWriters simply specify the offered minimum rate at which the data will be updated. DataReaders specify the requested minimum rate at which they expect data to be updated by publishers. If any of the DEADLINE periods are not met, the applications are notified by the Listeners attached to the DataWriter or DataReader objects.

2.2.6.8 Resource Limits

RESOURCE_LIMITS enables applications to control their memory consumption. This QoS policy also serves as a hint to the middleware implementation so that necessary memory can be pre-allocated. Pre-allocated memory is generally a good practice in real-time systems when used correctly. It is more efficient and it prevents memory fragmentation which is a general problem of dynamic memory allocation.

2.2.6.9 Time Based Filter

TIME_BASED_FILTER can be used to limit the number of samples received for periodically published data. This is a good example of the decoupling between publishers and subscribers offered by DDS middleware. Regardless of the publisher's transmission rate the subscribers can limit the rate by specifying an inter-message time limit. The middleware then guarantees that samples are not delivered faster than the limit by discarding samples.

CHAPTER 3

DDS BASED MIL-STD-1553B SIMULATION IMPLEMENTATION

This chapter discusses the design and the implementation details of the DDS based 1553 data bus interface simulation.

3.1 Development Environment

The simulation tool is developed in C++. For portability and maintenance reasons, all of the dependencies are selected to be portable across operating systems. The main development environment is the Ubuntu 11.10 64bit operating system. An open source DDS implementation is used as the middleware, namely OpenSplice DDS. In addition "boost" C++ libraries are used as an adaptation layer for threading and other operating system dependent system calls. Finally, for XML parsing purposes the widely available Xerces-C library is used.

3.2 Design

MIL-STD-1553B does not define a standard programming interface, so every vendor provides its own API for programming the 1553 cards. This causes problems when trying to port existing software components on a different vendor's card. The simulation tool implemented in this thesis also has its own API, but it was designed with object oriented design patterns so that the users can easily switch between simulation and real hardware cards without substantial changes in their source codes. This was achieved with a combination of Factory and Strategy design patterns [21].

Figure 3.1 shows the public API and some of the key private classes that are important for the implementation of the simulation library. The following sections discusses those classes and their methods.

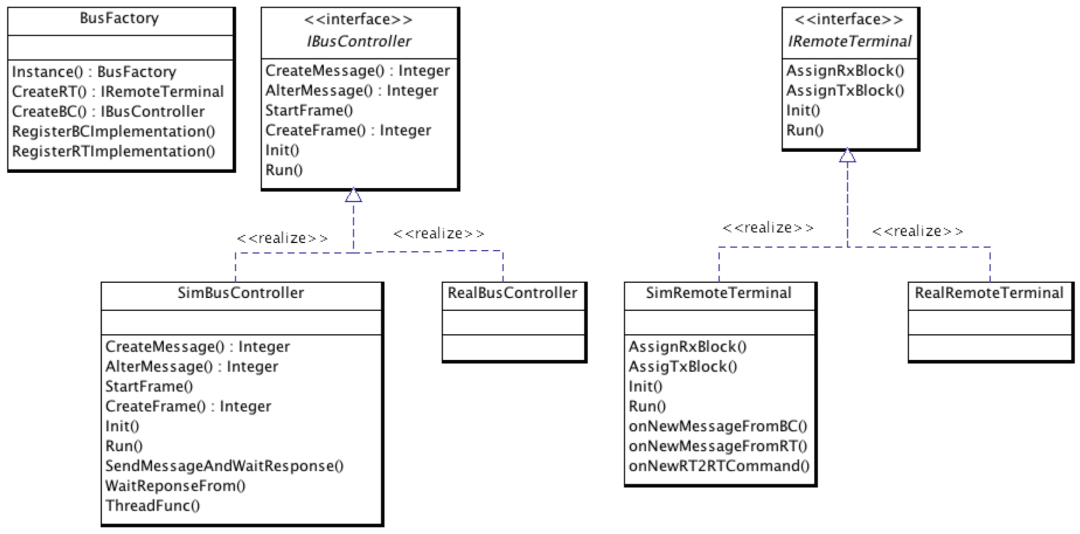


Figure 3.1: Class Diagram of the 1553 data bus simulation library.

3.2.1 BusFactory (public)

BusFactory is a singleton class implemented as the entry point of the simulation library. It acts as the factory of BusController and RemoteTerminal objects, with its createBC and createRT methods. Using this pattern, it is possible to implement a single programming interface which can abstract both the simulation library and the real hardware programming interface.

3.2.1.1 Instance

This method creates or returns the single instance of the BusFactory class using the Singleton pattern.

3.2.1.2 CreateBC

This method creates a new BusController object. By default an instance of SimBusController is returned. The users of the library may choose to implement their own BusController implementations and register to the factory with RegisterBCImplementation method.

3.2.1.3 CreateRT

This method creates a new RemoteTerminal object. By default an instance of SimRemoteTerminal is returned. The users of the library may choose to implement their own

RemoteTerminal implementations and register to the factory with the RegisterRTImplementation method.

3.2.1.4 RegisterBCImplementation

This method is used to register new BusController implementations to the library. The purpose of providing this method is to enable users to wrap their vendor specific 1553 card API under the BusController abstract interface. By using this design, the users are able to choose between the simulation library or real card without the need to modify their code or the simulation library code.

3.2.1.5 RegisterRTImplementation

This method is used to register new RemoteController implementations to the library. The purpose of this method is the same as the one described for the RegisterBCImplementation, but this time with focus on RemoteTerminal.

3.2.2 IBusController (public)

IBusController is an abstract class used to encapsulate the interface of a 1553 bus controller. It has all the necessary methods to be able to define commands, messages and frames.

3.2.2.1 CreateMessage

This method is used to create a 1553 message. An integer id is returned for the created message which can be later used to insert the message into a frame.

3.2.2.2 AlterMessage

This method is used to change the data of an already existing message. The type of the message remains unchanged and only the data is replaced. This is used generally used for periodic data that changes over time.

3.2.2.3 CreateFrame

This method is used to create a minor frame. The major frame consists of all the minor frames in the order of their creation time.

3.2.2.4 StartFrame

This method is used to set the starting minor frame for the major frame. By default the first created minor frame is transmitted first, but the users are able to change it with this method by providing a minor frame id.

3.2.2.5 Run

This method is used to start the BusController operation which starts transmitting the messages that were defined in the major and minor frames.

3.2.3 IRemoteTerminal (public)

IRemoteTerminal is an abstract class used to encapsulate the interface of a 1553 remote terminal. It has all the necessary methods to be able to define receive and transmit blocks in the 32 subaddresses it manages.

3.2.3.1 AssignRxBlock

This method is used to assign a receive block that is related with a subaddress. The messages that have been sent to the specified subaddress are received at this assigned block.

3.2.3.2 AssignTxBlock

This method is used to assign a transmit block that is related with a subaddress. A data is assigned to this block for transmission. The data is transmitted when a command for transmission is received targeted at the specified subaddress.

3.2.3.3 Run

This method is used to star the RemoteTerminal operation. The RemoteTerminal starts listening to commands and messages and responds accordingly.

3.2.4 SimBusController (private)

SimBusController class is an implementation of the IBusController interface, which actually contains the simulation logic of a 1553 card that is programmed as the bus controller. This class is not available in the public interface of the simulation library. The users create an IBusController object from the BusFactory which creates an instance of this class and returns it. The users use the IBusController interface. This class uses DDS to send commands to remote terminals and also asynchronously waits for responses from remote terminals again over DDS.

3.2.5 SimRemoteTerminal (private)

SimRemoteTerminal class is an implementation of the IRemoteTerminal interface, which actually contains the simulation logic of a 1553 card thats programmed as a remote terminal. This class is not available in the public interface of the simulation library. The users create an IRemoteTerminal object from the BusFactory which creates and instance of this class and returns it. The users use the IRemoteTerminal interface. This class uses DDS to asynchronously wait for commands from the bus controller and immediately responds to the commands with DDS.

3.3 DDS Data Types

According to the DDS specification, IDL is used to define the data types. Listing 3.1 shows the DDS types defined for the implementation of the simulation library. All types have a corresponding DDS Topic with the same name.

Listing 3.1: The definitions of DDS Types used in the simulation library.

```
module DataTypes
{
    struct MessageFromBC {
        short rtAddr;
        short subAddr;
        short type;
        sequence<short, 32> data;
    };
    #pragma keylist MessageFromBC

    struct RT2RTCommand {
        short receiverRtAddr;
        short receiverSubAddr;
        short transmitterRtAddr;
        short transmitterSubAddr;
    };
    #pragma keylist RT2RTCommand

    struct MessageFromRT {
        short sourceRtAddr;
        short subAddr;
        sequence<short, 32> data;
    };
    #pragma keylist MessageFromRT
};
```

MessageFromBC is used to send messages from bus controller to remote terminals. It contains the remote terminal address and the subaddress within the target terminal. Additionally this Topic can carry a maximum of thirty-two data words as described in MIL-STD-1553B.

RT2RTCommand is defined to handle the initiation of a remote terminal to remote terminal (RT2RT) transaction. This Topic is published by the bus controller whenever such an RT2RT transaction is needed. Remote terminals will listen to this Topic and either transmit data or start listening for data from the other remote terminal.

MessageFromRT Topic is published by the remote terminals as a response to commands from bus controllers. This is actually implementing the status word described in MIL-STD-1553B with omissions for simplicity of the implementation. This Topic can also contain a maximum of thirty-two data words to be sent in response to transmit commands.

3.4 Advantages of Using DDS in the Simulation Tool

This section describes some features of DDS middleware that are used in the implementation of the simulation tool. The advantages of using these features are also discussed.

3.4.1 Publish-Subscribe and Automatic Discovery

DDS middleware provides automatic discovery of publishers and subscribers without the need to specify any direct addressing. The users just define their Topics and create DataWriters for publishing and DataReaders for subscribing. All the negotiations to discover and connect those reader and writers is handled by the DDS middleware transparently to the user. This provides a great flexibility of deploying the simulation tool. Without the need for configuration, the simulated bus controller and remote terminals can be deployed to run on multiple computers and if desired all of them can run on a single computer.

3.4.2 Content Filtering

The 1553 data bus hardware and software components perform filtering according to the addresses specified in the commands. Only the remote terminal that is targeted with a command receives and processes the message and all other remote terminals filter and discard the message. The same behaviour is emulated in the simulation library with the content filtering features provided by DDS middleware.

Each SimRemoteTerminal object creates a ContentFilteredTopic for both MessageFromBC and RT2RTCommands to receive only the messages that are addressed to itself. DDS middleware handles the filtering and discards messages that are not addressed to the specified remote terminal. Content filtering capabilities of DDS simplify the simulation tool implementation since there's no need for conditional checks on the application layer. Moreover, DDS middleware preserves bandwidth by applying the filters on the publisher side.

3.4.3 Command/Response with WaitSet

DDS is, by definition, an asynchronous middleware. The publishers do not block and wait for response after a data is sent and the subscribers are not required to block and wait on data arrival. Subscribers are asynchronously notified when a data is published and is available for consumption. But DDS also provides a mechanism to implement synchronous operations. The WaitSet class with the help of Condition classes can be used to implement this behaviour.

The 1553 data bus protocol has a synchronous command/response nature when the bus controller sends a command and waits the status messages back from the remote terminals. In order to implement this behaviour the WaitSet and QueryCondition classes of DDS were used. QueryCondition is a specialised Condition class that can be constructed with an SQL like query. The QueryCondition object is attached to the WaitSet and the "wait" operation is called which blocks the calling thread until a data arrives which satisfies the query condition. A timeout can also be specified for the blocking wait. This timeout mechanism is used to implement the "Response Time" timing requirement defined in MIL-STD-1553B.

The same synchronous behavior can be implemented by using the native conditional variables of the operating system. However, using the DDS provided WaitSet mechanism makes the implementation both portable and maintainable. In addition, the implementation of simulation tool is simplified since there's no need for doing conditional checks.

3.5 Deployment

The simulation tool consists of a library and two executable files, namely busController and remoteTerminal. The library can be used to implement bus controller and remote terminal interfaces with C++. The executables are provided for the purpose of loading and running scenarios defined with XML files. Figure 3.2 shows an example deployment of a simulated 1553 environment. The simulation can run on Ethernet networks with the help of DDS library. Since DDS is capable of handling multiple computers transparently by automatically discovering publishers and subscribers each remote terminal can be run on a different computer on the network. It is also possible to run bus controller and each remote terminal on the same computer. In that case depending on the DDS implementation either shared memory or the loopback network interfaces are used for the data transfers.

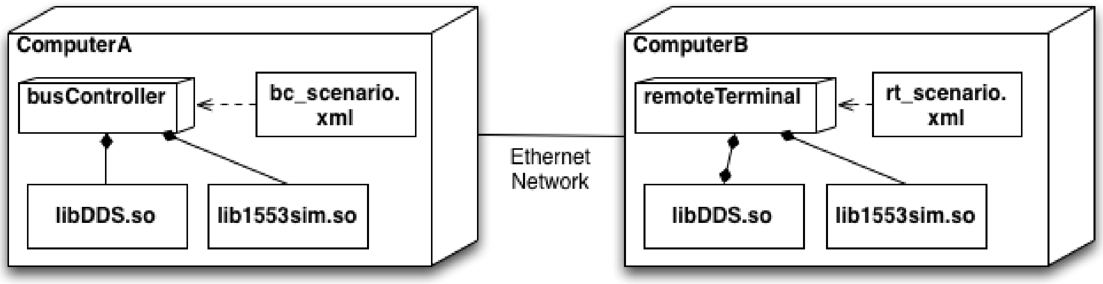


Figure 3.2: Example Deployment of a bus controller and a single remote terminal on different computers.

3.6 XML Scenario Interface

In addition to the C++ programming interfaces, the simulation library also supports an easier scenario definition interface with XML files. The XML schemas are designed to mimic the C++ interface so the system designers can be familiar with both interfaces.

There are mainly two XML schemas, one for the bus controller and one for the remote terminal scenario definitions. Each remote terminal shall define its own scenario in a separate XML file. The simulation tool provides executables to load and run the scenarios defined by those XML files.

The bus controller XML schema can be seen in Figure 3.3. According to this schema, a root "bus_controller" element contains two child elements, "messages" and "stages". The "messages" element is where all the message formats used in the bus controller are defined. The "stages" element contains different major frames to be run in stages, for example initialization stage and operational stages. The bus controller simply runs the major frames according to the order they are defined in the XML file. See Appendix C for the actual XML schema and Appendix A for an example XML file compliant with the schema.

The remote terminal XML schema can be seen in Figure 3.4. This schema is simpler because all of the message formats, major and minor frames are defined in bus controller. The remote terminal contains a root "remote_terminal" element which has an "id" attribute. This "id" corresponds to the remote terminal address which can be at most 31. There are two child elements to assign data for transmission from a specific subaddress or to assign a subaddress for data reception. See Appendix D for the actual XML schema and Appendix B for an example XML file compliant with the schema.

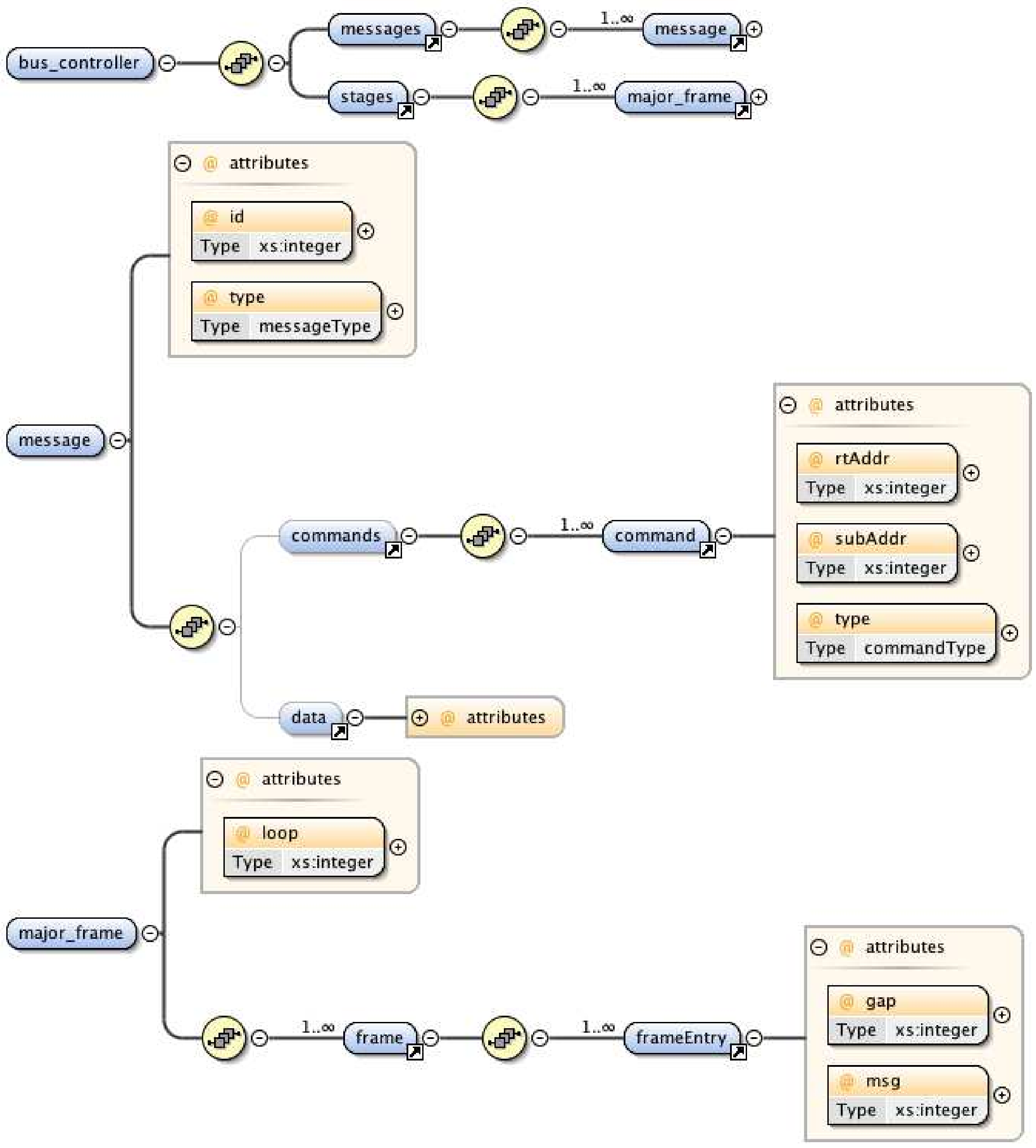


Figure 3.3: The XML Schema for the Bus Controller XML Files.

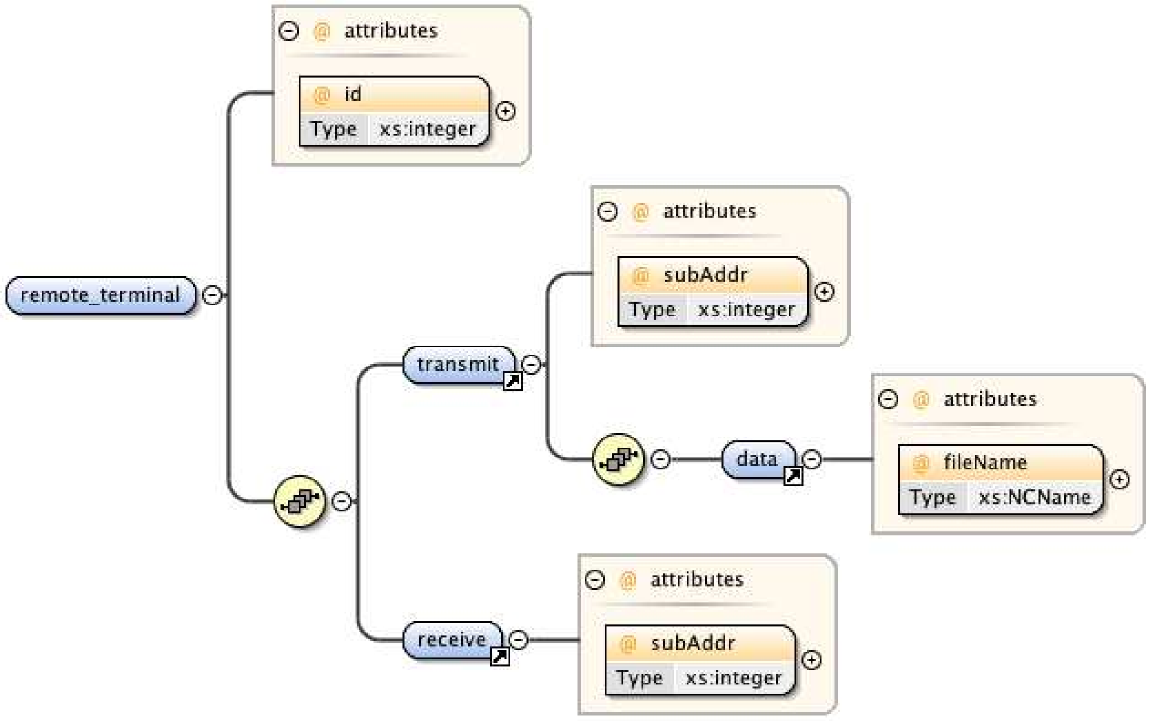


Figure 3.4: The XML Schema for the Remote Terminal XML Files.

CHAPTER 4

CASE STUDY: A BASIC MIL-STD-1760 SIMULATION

MIL-STD-1760, Aircraft/Store Electrical Interconnection System (AEIS), standard published by the USA Department of Defense defines electrical interfaces between military aircraft and its stores. Stores include, but are not limited to, weapons, fuel tanks, pods and buoys. The purpose of the standard is to enhance the interoperability between stores and aircraft by defining specific electrical/optical, logical, and physical requirements for the AEIS [22]. The standard requires data communications between the aircraft and its stores to be carried over the MIL-STD-1553B data bus.

In this thesis, a case study of a basic MIL-STD-1760 simulation was performed. This gives the best overview of how the simulation tool performs on the realistic scenarios that conform to the MIL-STD-1760 standard. For the purpose of easy generation of XML scenario definition files a modeling tool was developed [9]. The tool provides a spreadsheet based graphical interface to fill data for the MIL-STD-1760 specific messages and then generate the XML files defined for the 1553 data bus simulation.

The tool generates the following 1760 messages:

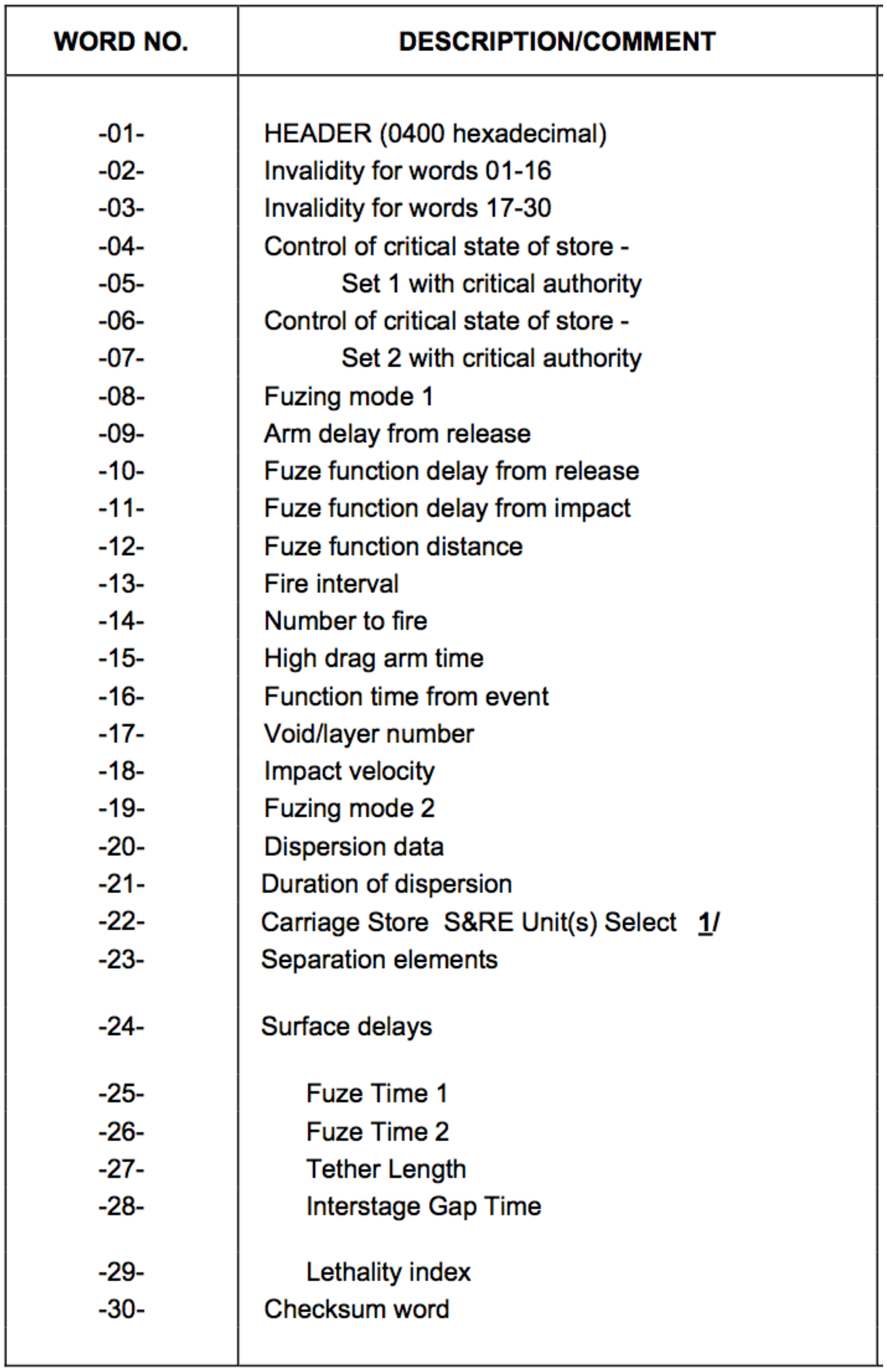
- **Store Control:** This standard message is used to control the state of stores. It is sent from the aircraft (bus controller) and contains a receive command and thirty data words that is sent to the subaddress 11 (00111) of the store (remote terminal). See Figure 4.1.
- **Store Monitor:** This message is a status message transmitted from the store (remote terminal). It contains thirty data words sent from the subaddress 11 (00111). See Figure 4.2.
- **Store Description:** This message contains the identity of the store. It contains

thirty data words sent from the store to the aircraft from the subaddress 1 (00001). See Figure 4.3.

- **Aircraft Description:** This message contains the identity of the aircraft. It contains thirty data words sent from the aircraft to the store to the subaddress 1 (00001). See Figure 4.4.

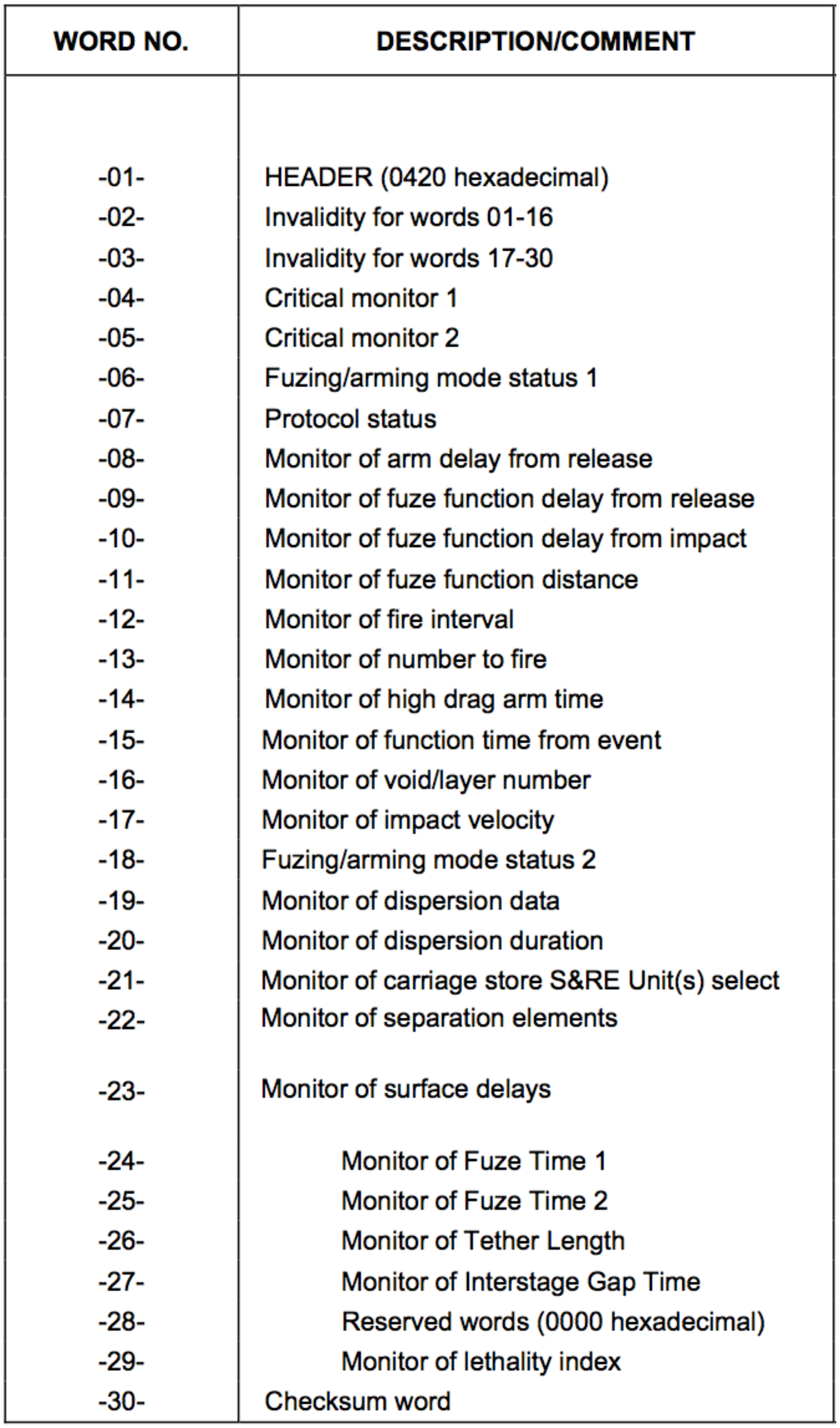
In this case study, a simulation of a power up sequence of two imaginary missiles and an aircraft is performed. MIL-STD-1760 is utilized to design this fictitious power up sequence. One can refer to the standard for the details of the mentioned messages.

The scenario is illustrated in Figure 4.5. The scenario is triggered when the aircraft applies 28V DC1 to the missile interfaces. Than say that missiles start to provide valid responses to the aircraft within 50 ms. For the first 200 ms missiles provide a valid status word with a busy condition to Store Description requests. After 200 ms, missiles send initial Store Description message with a valid data word. As soon as initial valid Store Description is received, aircraft sends Store Control messages for controlling the state of stores and initiates a built-in test. Missiles start transmitting Store Monitor messages with a period of 25 Hz. Store Monitor messages reflect the condition of the missiles. Each missile is expected to complete the built-in test within 4 seconds of receiving Store Control message. After the built-in test is completed, aircraft sends Aircraft Description message to each missile. The XML file for the bus controller (aircraft), remote terminal 1 (missile 1) and remote terminal 2 (missile 2) can be found in Appendix E, Appendix F and Appendix G respectively.



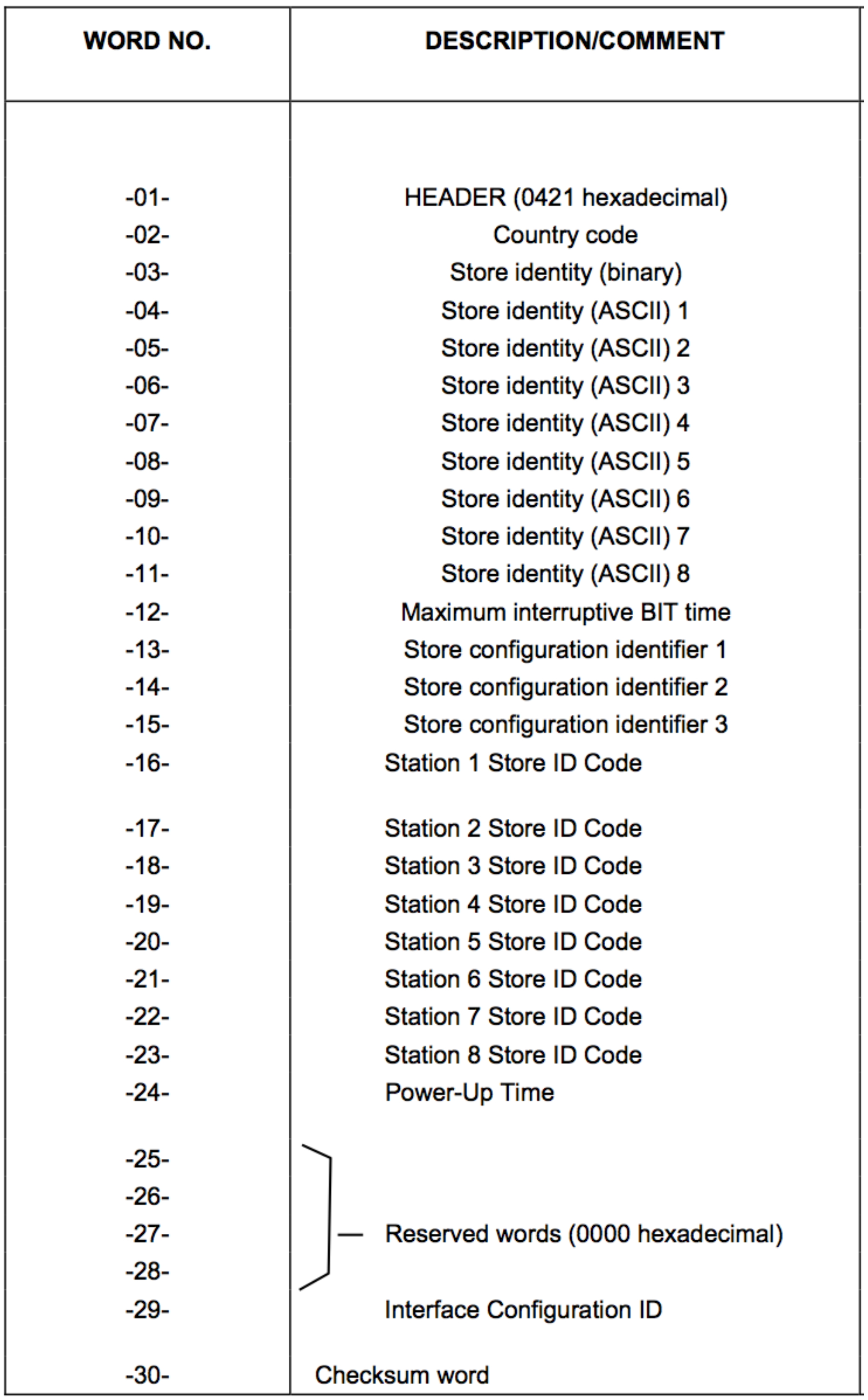
WORD NO.	DESCRIPTION/COMMENT
-01-	HEADER (0400 hexadecimal)
-02-	Invalidity for words 01-16
-03-	Invalidity for words 17-30
-04-	Control of critical state of store -
-05-	Set 1 with critical authority
-06-	Control of critical state of store -
-07-	Set 2 with critical authority
-08-	Fuzing mode 1
-09-	Arm delay from release
-10-	Fuze function delay from release
-11-	Fuze function delay from impact
-12-	Fuze function distance
-13-	Fire interval
-14-	Number to fire
-15-	High drag arm time
-16-	Function time from event
-17-	Void/layer number
-18-	Impact velocity
-19-	Fuzing mode 2
-20-	Dispersion data
-21-	Duration of dispersion
-22-	Carriage Store S&RE Unit(s) Select 1/
-23-	Separation elements
-24-	Surface delays
-25-	Fuze Time 1
-26-	Fuze Time 2
-27-	Tether Length
-28-	Interstage Gap Time
-29-	Lethality index
-30-	Checksum word

Figure 4.1: Store Control message defined in MIL-STD-1760.



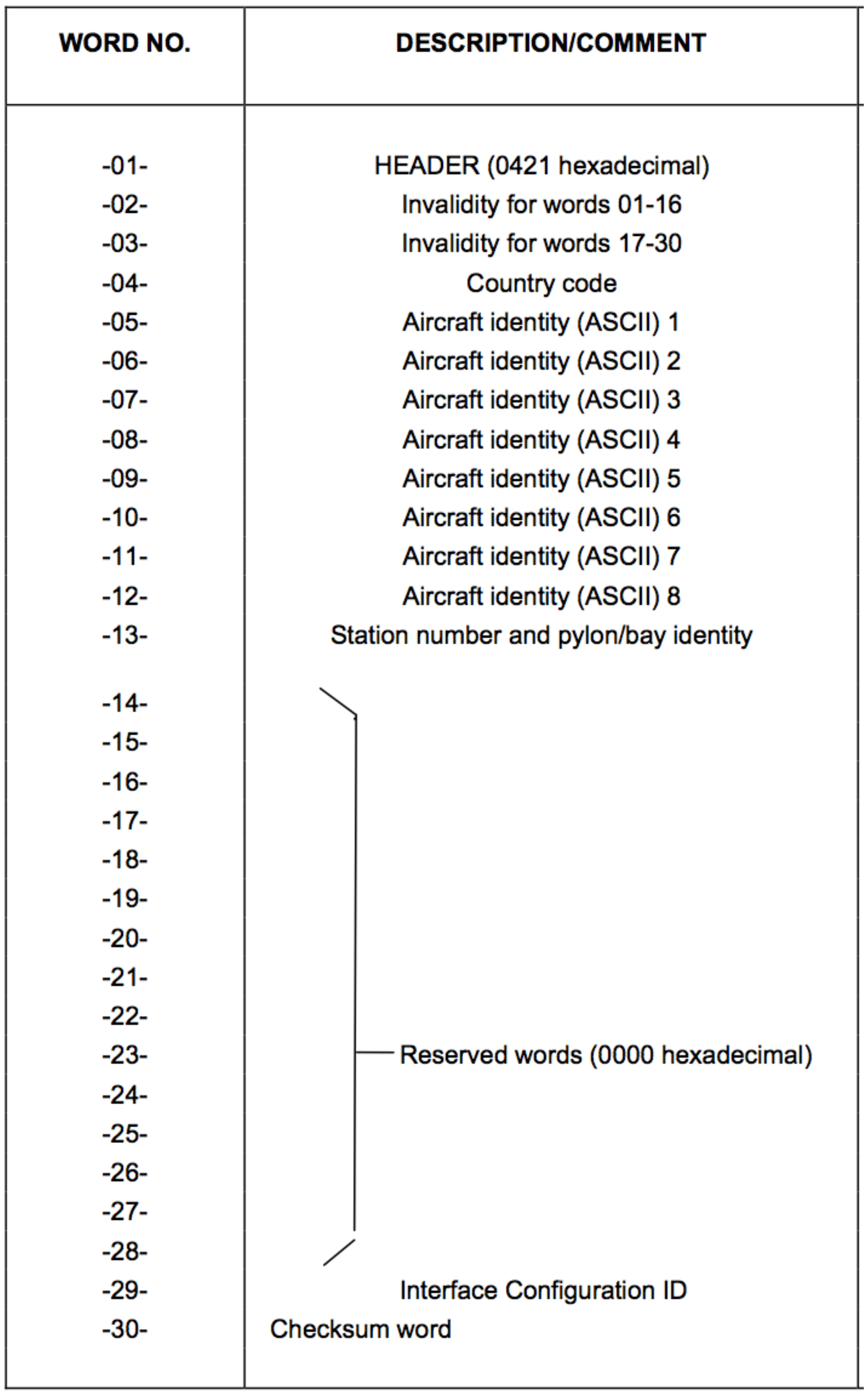
WORD NO.	DESCRIPTION/COMMENT
-01-	HEADER (0420 hexadecimal)
-02-	Invalidity for words 01-16
-03-	Invalidity for words 17-30
-04-	Critical monitor 1
-05-	Critical monitor 2
-06-	Fuzing/arming mode status 1
-07-	Protocol status
-08-	Monitor of arm delay from release
-09-	Monitor of fuze function delay from release
-10-	Monitor of fuze function delay from impact
-11-	Monitor of fuze function distance
-12-	Monitor of fire interval
-13-	Monitor of number to fire
-14-	Monitor of high drag arm time
-15-	Monitor of function time from event
-16-	Monitor of void/layer number
-17-	Monitor of impact velocity
-18-	Fuzing/arming mode status 2
-19-	Monitor of dispersion data
-20-	Monitor of dispersion duration
-21-	Monitor of carriage store S&RE Unit(s) select
-22-	Monitor of separation elements
-23-	Monitor of surface delays
-24-	Monitor of Fuze Time 1
-25-	Monitor of Fuze Time 2
-26-	Monitor of Tether Length
-27-	Monitor of Interstage Gap Time
-28-	Reserved words (0000 hexadecimal)
-29-	Monitor of lethality index
-30-	Checksum word

Figure 4.2: Store Monitor message defined in MIL-STD-1760.



WORD NO.	DESCRIPTION/COMMENT
-01-	HEADER (0421 hexadecimal)
-02-	Country code
-03-	Store identity (binary)
-04-	Store identity (ASCII) 1
-05-	Store identity (ASCII) 2
-06-	Store identity (ASCII) 3
-07-	Store identity (ASCII) 4
-08-	Store identity (ASCII) 5
-09-	Store identity (ASCII) 6
-10-	Store identity (ASCII) 7
-11-	Store identity (ASCII) 8
-12-	Maximum interruptive BIT time
-13-	Store configuration identifier 1
-14-	Store configuration identifier 2
-15-	Store configuration identifier 3
-16-	Station 1 Store ID Code
-17-	Station 2 Store ID Code
-18-	Station 3 Store ID Code
-19-	Station 4 Store ID Code
-20-	Station 5 Store ID Code
-21-	Station 6 Store ID Code
-22-	Station 7 Store ID Code
-23-	Station 8 Store ID Code
-24-	Power-Up Time
-25-	
-26-	
-27-	— Reserved words (0000 hexadecimal)
-28-	
-29-	Interface Configuration ID
-30-	Checksum word

Figure 4.3: Store Description message defined in MIL-STD-1760.



WORD NO.	DESCRIPTION/COMMENT
-01-	HEADER (0421 hexadecimal)
-02-	Invalidity for words 01-16
-03-	Invalidity for words 17-30
-04-	Country code
-05-	Aircraft identity (ASCII) 1
-06-	Aircraft identity (ASCII) 2
-07-	Aircraft identity (ASCII) 3
-08-	Aircraft identity (ASCII) 4
-09-	Aircraft identity (ASCII) 5
-10-	Aircraft identity (ASCII) 6
-11-	Aircraft identity (ASCII) 7
-12-	Aircraft identity (ASCII) 8
-13-	Station number and pylon/bay identity
-14- to -28-	Reserved words (0000 hexadecimal)
-29-	Interface Configuration ID
-30-	Checksum word

Figure 4.4: Aircraft Description message defined in MIL-STD-1760.

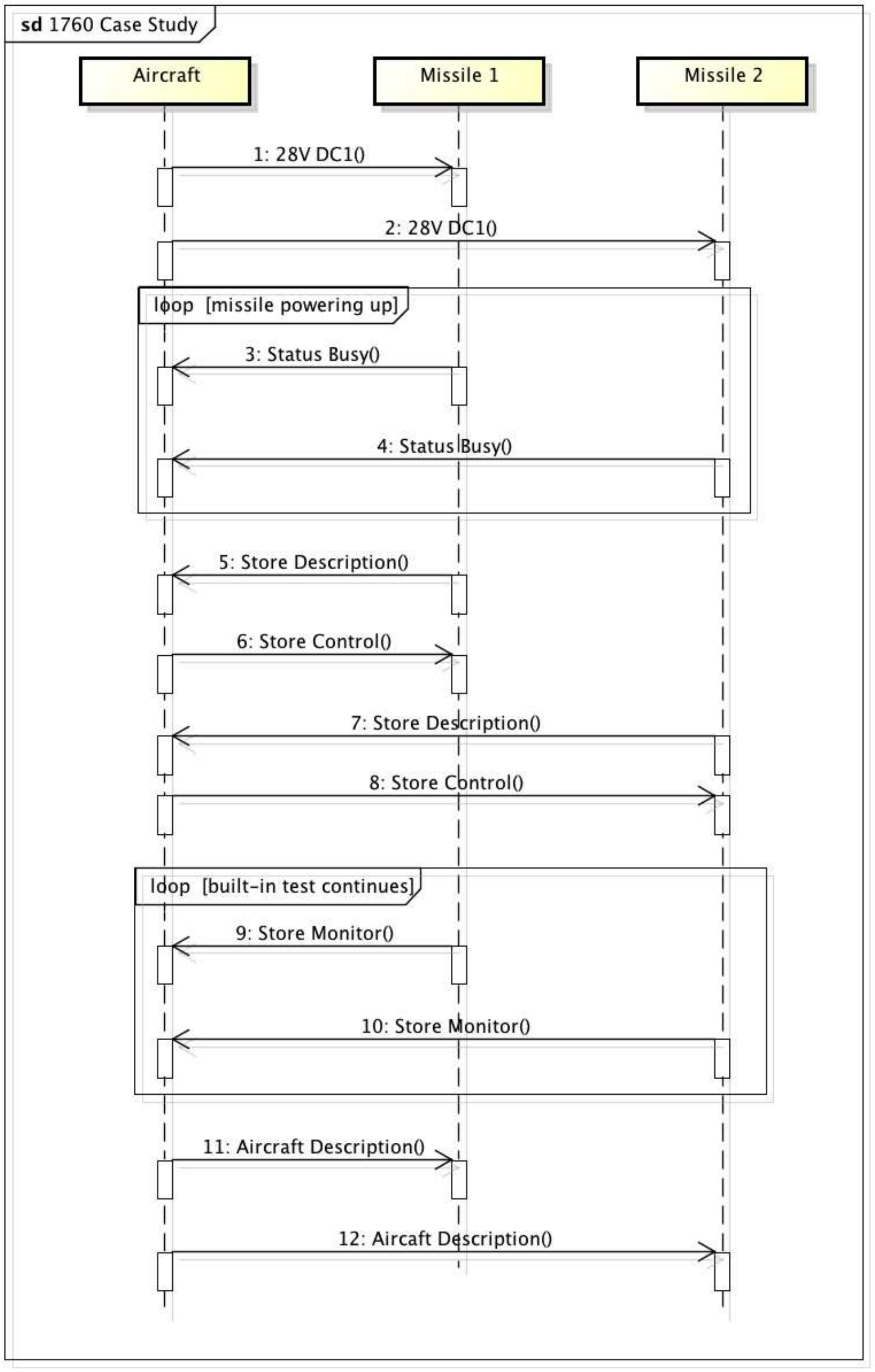


Figure 4.5: Sequence diagram of the MIL-STD-1760 scenario.

CHAPTER 5

PERFORMANCE ANALYSIS

MIL-STD-1553B has some strict timing requirements that were discussed at subsection 2.1.5. This thesis does not have a purpose of satisfying those requirements since for implementing them we would need to run the simulation on a real-time operating system. This contradicts with the purpose of this thesis which is to make the simulation tool easily accessible on widely available operating systems and networking infrastructures with minimum costs. However, some performance tests were performed to see what kind of timing limitations we have with the simulation tool. The performance test results of the middleware are also useful for some time related parameters that were used in the simulation.

The aim of the performance analysis was to see what kind of latency does the simulation tool have on a general purpose personal computer. The performance tests were performed on a single computer. See Table Table 5.1 for the actual test setup information.

Table 5.1: Performance Test Setup

Processor	Intel Core2 Quad Core Q9500 2.83 Ghz
Memory	6GB RAM
Operating System	Ubuntu 11.10 64bit
DDS Implementation	OpenSplice v5.4.1 OSS

With this setup basically three tests were performed:

- **BC2RT Latency:** This is the latency of a bus controller to remote terminal message with thirty-two data words. The latency is measured from the start of sending the command with data until the status message is received by the bus controller. See Figure 5.1.

- **RT2BC Latency:** This is the latency of a remote terminal to bus controller message with thirty-two data words. The latency is measured from the start of sending the transmit command to remote terminal until the status and data words are received on the bus controller. See Figure 5.2.

- **RT2RT Latency:** This is the latency of a remote terminal to remote terminal message with thirty-two data words. The latency is measured from the start of sending the receive and transmit command words to both remote terminals, until the status message from the receiving terminal arrives on the bus controller. See Figure 5.3.

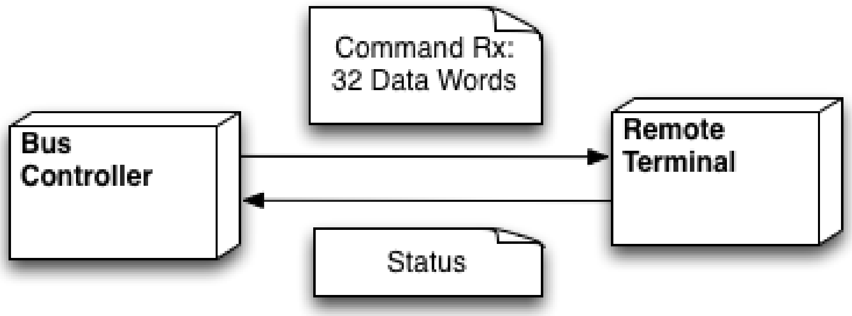


Figure 5.1: Performance Test of a BC2RT message on the 1553 data bus simulation tool.

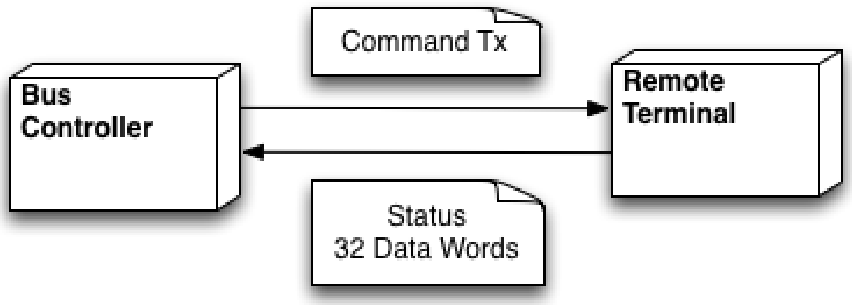


Figure 5.2: Performance Test of a RT2BC message on the 1553 data bus simulation tool.

Each performance test was performed for 100 times and the average latency was calculated. For a better comparison with the latencies of a real 1553 data bus the tests were also

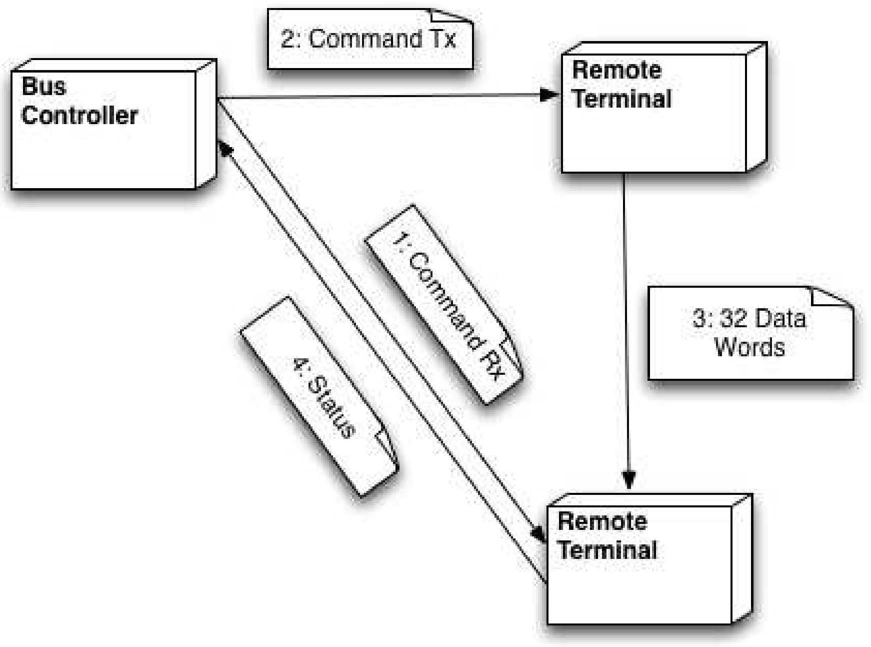


Figure 5.3: Performance Test of a RT2RT message on the 1553 data bus simulation tool.

run with messages containing only one data word. The results showed that the simulation tool can consistently provide latencies in microseconds range. See Table 5.2 for the results.

Table 5.2: Performance Test Results

Message Format	Mean Latency	Variance	Standard Deviation
BC2RT (32 Data Words)	193 μs	3852 μs	62 μs
RT2BC (32 Data Words)	173 μs	3264 μs	57 μs
RT2RT (32 Data Words)	273 μs	7112 μs	84 μs
BC2RT (1 Data Word)	168 μs	1712 μs	41 μs
RT2BC (1 Data Word)	172 μs	523 μs	22 μs
RT2RT (1 Data Word)	240 μs	1570 μs	39 μs

The latencies on a real 1553 data bus can be calculated as shown on Table 5.3 since each of the command, status and data words have 20 bits and require 20 μs time to be transmitted on the bus. The inter-message gap time and response times are ignored for simplicity. Even without adding those times it can be seen that the DDS based 1553 simulation tool consistently provides lower latencies when thirty-two data words are transmitted. However, when only one data word is transmitted the simulation tool has higher latencies. Most of the MIL-STD-1553B scenarios use messages with multiple data words to be able to utilize the limited bandwidth of the bus. With these results it can be said that the simulation tool is suitable for simulating MIL-STD-1553B scenarios, especially scenarios with messages containing multiple data words, but system designers shall not expect the simulation to run within the strict timing requirements of the MIL-STD-1553B standard.

Table 5.3: Message Latencies on 1553 data bus

Message Format	Message Content	Latency on 1553
BC2RT (32 Data Words)	Command + 32 * Data + Status	680 μs
RT2BC (32 Data Words)	Command + Status + 32 * Data	680 μs
RT2RT (32 Data Words)	Command + Command + Status + 32 * Data + Status	720 μs
BC2RT (1 Data Word)	Command + Data + Status	60 μs
RT2BC (1 Data Word)	Command + Status + Data	60 μs
RT2RT (1 Data Word)	Command + Command + Status + Data + Status	100 μs

CHAPTER 6

CONCLUSIONS

This thesis describes a MIL-STD-1553B serial data bus interface and protocol simulation tool which uses the DDS middleware standard as the communication technology. The simulation tool enables easy and quick verification of component interfaces in the development environments. There's no need for expensive hardware and special wiring to be able to test the designed MIL-STD-1553B interfaces. Moreover, an XML based scenario definition model is designed so that interfaces can be easily defined and simulated without the need to program the with the API of the simulation library. The simulation tool is verified using the XML scenario model with a case study of a MIL-STD-1760 interface simulation. It is shown that the tool is able to simulate MIL-STD-1760 interfaces succesfully.

In this thesis, using the DDS middleware enabled a flexible distributed simulation of the 1553 data bus by providing automatic discovery of terminals without the need of configuration. The publish-subscribe communications between bus controller and remote terminals and the filtering capabilities of the DDS middleware simplifies the simulation logic and preserve network bandwidth. In addition, the DDS middleware provides an abstraction layer over the networking API of the operating system which enables a portable and maintainable implementation.

The simulation tool is developed as a proof of concept implementation, so for simplicity some of the messages defined in MIL-STD-1553B are excluded. Specifically the mode commands and the details of the status words are not implemented. The status words are only used as an indication that the command is received and processed by the remote terminal. Also, the simulation tool does not guarantee all the strict timing requirements of the MIL-STD-1553B standard, but it is shown with performance tests that it can consistently perform with microsecond latencies.

For future work, it may be possible to transform the simulation tool into an adaptation layer for the legacy MIL-STD-1553B system components into the next generation of communication systems. The DDS standard is highly capable of real-time and safety-critical communications. Instead of using DDS just for the simulation, it can be used as an abstraction layer of the next generation avionics network infrastructures. D. Parish et al [15] defined an emulation of the 1553 data bus on ATM networks so that they can port legacy components into a new ATM based communication system. A similar approach may be applied to the DDS based simulation tool on the AFDX (Avionics Full-Duplex Switched Ethernet) standard. AFDX is an emerging standard that defines a safety-critical high performance Ethernet based network infrastructure [23]. The new system components may use DDS based interfaces over the AFDX and the legacy components can be migrated to the new networking infrastructure using the simulation tool. Figure 6.1 illustrates this idea and shows the integration of legacy MIL-STD-1553B components into a next generation AFDX network. This approach would eliminate the wiring required for MIL-STD-1553B and reduce the weight on the aircraft.

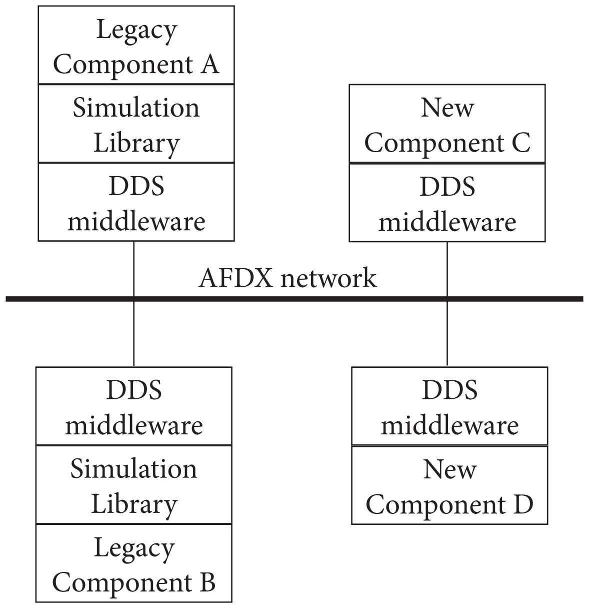


Figure 6.1: Integration of legacy 1553 components into an AFDX based network.

REFERENCES

[1] Condor Engineering, Inc., *MIL-STD-1553 Tutorial*, June 2000.

[2] Aziz, S. M., "A cycle-accurate transaction level SystemC model for a serial communication bus," *Comput. Electr. Eng.*, Vol. 35, No. 5, Sept. 2009, pp. 790–802.

[3] US Department Of Defense, *Aircraft Internal Time Division Command/Response Multiplex Data Bus*, 1978.

[4] *MIL-STD-1553 Designer's Guide*, ILC Data Device Corporation, 6th ed., 2003.

[5] Object Management Group (OMG), *Data Distribution Services for Real-time Systems Version 1.2*, 2007.

[6] Pardo-Castellote, G., Farabaugh, B., and Warren, R., "An Introduction to DDS and Data-Centric Communications," `http://www.omg.org/news/whitepapers/Intro_To_DDS.pdf`, August 2005, [Accessed: 20 August, 2012].

[7] Kutluca, H., Cetin, İ. E., Deniz, E., and Bal, B., "MilSOFT DDS Arakatmanı ve DDS'in Savaş Yönetim Sistemlerinde Simülasyon Amaçlı Kullanımı," *USMOS*, Ankara, Turkey, 2007.

[8] Downing, N., "Virtual MIL-STD-1553," *25th Digital Avionics Systems Conference, 2006 IEEE/AIAA*, oct. 2006, pp. 1 –8.

[9] Güçlü, K., "Testing a MIL-STD-1553 Bus Simulator with MIL-STD-1760 Based Test Scenarios," Term project report, Department of Computer Engineering, METU, 2012.

[10] "Electronic Warfare and Radar Systems Engineering Handbook," Tech. Rep. TP-8347, Avionics Department of the Naval Air Warfare Center Weapons Division, Washington, DC 20361, April 1999.

[11] Tian, J., Hu, K., Zhang, H., Niu, J., and Jiang, H., "Design of MIL-STD-1553B protocol simulation system," *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, Vol. 6, aug. 2010, pp. V6–389 –V6–392.

[12] Engblom, J. and Holm, C. M., "A Fully Virtual Multi-Node 1553 Bus Computer System," *Data Systems in Aerospace*, DASIA, May 2006.

[13] IEEE, "IEEE Standard for Standard SystemC Language Reference Manual," *IEEE Std 1666-2011 (Revision of IEEE Std 1666-2005)*, 9 2012, pp. 1 –638.

[14] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hållberg, G., Högberg, J., Larsson, F., Moestedt, A., and Werner, B., "Simics: A Full System Simulation Platform," *Computer*, Vol. 35, No. 2, Feb. 2002, pp. 50–58.

[15] Parish, D., Briggs, R., Chambers, D., Hunter, C., and Kelsall, N., "1553 emulation over ATM (asynchronous transfer mode) - A hybrid avionics communications architecture," *Aerospace and Electronic Systems Magazine, IEEE*, Vol. 13, No. 3, march 1998, pp. 34 –39.

[16] Alta Data Technologies, *MIL-STD-1553 Tutorial and Reference*, 2007.

[17] Naval Surface Warfare Center Dahlgren Division, *Open Architecture (OA) Computing Environment Design Guidance*, August 2004.

[18] Kutluca, H., Cetin, İ. E., Deniz, E., Bal, B., Kılıç, M., and Çakır, U., "Developing MilSOFT DDS Middleware," *OMG Real-time and Embedded Systems Workshop*, Arlington-VA USA, July 2007.

[19] Object Management Group (OMG), *The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification, Version 2.1*, November 2010.

[20] Object Management Group (OMG), *Interface Definition Language (IDL) Specification, Version 3.5*, 2011.

[21] Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. M., "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP '93, Springer-Verlag, London, UK, UK, 1993, pp. 406–431.

[22] US Department of Defense, *MIL-STD-1760E Aircraft/Store Electrical Interconnection System*, October 2007.

[23] TechSAT, "AFDX / ARINC 664 Tutorial," `http://www.techsat.com/fileadmin/media/pdf/infokiosk/TechSAT_TUT-AFDX-EN.pdf`, 2008, [Accessed: 30 August, 2012].

[24] Pardo-Castellote, G., "OMG Data-Distribution Service: Architectural Overview," *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCSW '03, IEEE Computer Society, Washington, DC, USA, 2003, pp. 200–.

[25] Corsaro, A., Querzoni, L., Scipioni, S., Piergiovanni, S. T., and Virgillito, A., "Quality of Service in Publish/Subscribe," *Global Data Management*, IOS Press, 2006.

[26] MilesTek, "Mil-STD-1553B Concepts and Considirations," `http://www.milestek1553.com/tech/PDF/MTI-1553B-40.pdf`, 2012, [Accessed: 30 August, 2012].

Appendix A

EXAMPLE BUS CONTROLLER XML FILE

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<bus_controller xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
  xsi:noNamespaceSchemaLocation="bc_schema.xsd">
     <messages>
          <message id="1" type="BC2RT">
               <commands>
                    <command rtAddr="1" subAddr="1" type="receive" />
               </commands>
               <data fileName="track1.xml" />
          </message>
          <message id="2" type="RT2RT">
               <commands>
                    <command rtAddr="1" subAddr="1" type="receive" />
                    <command rtAddr="2" subAddr="2" type="transmit" />
               </commands>
          </message>
          <message id="3" type="RT2BC">
               <commands>
                    <command rtAddr="2" subAddr="1" type="transmit" />
               </commands>
          </message>
          <message id="5" type="BC2RT">
```

```
            <commands>
                <command rtAddr="2" subAddr="1" type="receive" />
            </commands>
            <data fileName="track2.xml" />
        </message>
    </messages>
    <stages>
        <major_frame loop="1">
            <frame>
                <frameEntry msg="1" gap="1000000" />
            </frame>
        </major_frame>
        <major_frame loop="0">
            <frame>
                <frameEntry msg="2" gap="1000000" />
                <frameEntry msg="4" gap="1000000" />
            </frame>
            <frame>
                <frameEntry msg="3" gap="1000000" />
                <frameEntry msg="4" gap="1000000" />
                <frameEntry msg="5" gap="1000000" />
            </frame>
        </major_frame>
    </stages>
</bus_controller>
```

Appendix B

EXAMPLE REMOTE TERMINAL XML FILE

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<remote_terminal xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
     xsi:noNamespaceSchemaLocation="rt_schema.xsd" id="1">
     <transmit subAddr="1">
          <data fileName="rtdata1.xml" />
     </transmit>
     <receive subAddr="2" />
</remote_terminal>
```

Appendix C

XML SCHEMA FOR BUS CONTROLLER

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- <xs:import namespace="http://www.w3.org/2001/XMLSchema-
      instance" schemaLocation="xsi.xsd"/> -->
  <xs:element name="bus_controller">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="messages"/>
        <xs:element ref="stages"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="message">
    <xs:complexType>
      <xs:sequence>
        <xs:element minOccurs="0" ref="commands"/>
        <xs:element minOccurs="0" ref="data"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:integer"/>
      <xs:attribute name="type" use="required" type="messageType"/
        >
    </xs:complexType>
  </xs:element>
```

```
<xs:element name="major_frame">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="frame"/>
    </xs:sequence>
    <xs:attribute name="loop" use="required" type="xs:integer"/>
  </xs:complexType>
</xs:element>
<xs:element name="messages">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="message"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="commands">
  <xs:complexType>
    <xs:sequence>
      <xs:element maxOccurs="unbounded" ref="command"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="command">
  <xs:complexType>
    <xs:attribute name="rtAddr" use="required" type="xs:integer"
       />
    <xs:attribute name="subAddr" use="required" type="xs:integer
       "/>
    <xs:attribute name="type" use="required" type="commandType"/
       >
  </xs:complexType>
</xs:element>
<xs:element name="data">
```

```
    <xs:complexType>
      <xs:attribute name="fileName" use="required" type="xs:NCName
         "/>
    </xs:complexType>
  </xs:element>
  <xs:element name="stages">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="major_frame"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="frame">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="frameEntry"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="frameEntry">
    <xs:complexType>
      <xs:attribute name="gap" use="required" type="xs:integer"/>
      <xs:attribute name="msg" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:simpleType name="commandType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="receive"/>
      <xs:enumeration value="transmit"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:simpleType name="messageType">
    <xs:restriction base="xs:string">
```

```
        <xs:enumeration value="BC2RT"/>
        <xs:enumeration value="RT2BC"/>
        <xs:enumeration value="RT2RT"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:schema>
```

Appendix D

XML SCHEMA FOR REMOTE TERMINALS

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
  <xs:element name="remote_terminal">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="transmit"/>
        <xs:element ref="receive"/>
      </xs:sequence>
      <xs:attribute name="id" use="required" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="transmit">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="data"/>
      </xs:sequence>
      <xs:attribute name="subAddr" use="required" type="xs:integer
          "/>
    </xs:complexType>
  </xs:element>
  <xs:element name="data">
    <xs:complexType>
```

```
      <xs:attribute name="fileName" use="required" type="xs:NCName
          "/>
    </xs:complexType>
  </xs:element>
  <xs:element name="receive">
    <xs:complexType>
      <xs:attribute name="subAddr" use="required" type="xs:integer
          "/>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Appendix E

CASE STUDY BUS CONTROLLER XML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<bus_controller xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
  xsi:noNamespaceSchemaLocation="bc_schema.xsd">
    <messages>
          <!-- messages for RT 1 -->
          <message id="1" type="RT2BC">
               <commands>
                    <command rtAddr="1" subAddr="1" type="transmit" />
               </commands>
          </message>
          <message id="2" type="BC2RT">
               <commands>
                    <command rtAddr="1" subAddr="3" type="receive" />
               </commands>
               <data fileName="scenarios/1760powerup/storeControl.xml
                  " />
          </message>
          <message id="3" type="BC2RT">
               <commands>
                    <command rtAddr="1" subAddr="4" type="receive" />
               </commands>
               <data fileName="scenarios/1760powerup/aircraftDesc.xml
```

```
            " />
    </message>
    <message id="4" type="RT2BC">
        <commands>
            <command rtAddr="1" subAddr="4" type="transmit" />
        </commands>
    </message>
    <!-- messages for RT 2 -->
    <message id="5" type="RT2BC">
        <commands>
            <command rtAddr="2" subAddr="1" type="transmit" />
        </commands>
    </message>
    <message id="6" type="BC2RT">
        <commands>
            <command rtAddr="2" subAddr="3" type="receive" />
        </commands>
        <data fileName="scenarios/1760powerup/storeControl.xml
            " />
    </message>
    <message id="7" type="BC2RT">
        <commands>
            <command rtAddr="2" subAddr="4" type="receive" />
        </commands>
        <data fileName="scenarios/1760powerup/aircraftDesc.xml
            " />
    </message>
    <message id="8" type="RT2BC">
        <commands>
            <command rtAddr="2" subAddr="4" type="transmit" />
        </commands>
    </message>
</messages>
```

```
    <stages>
        <major_frame loop="5">
                <frame>
                    <frameEntry msg="1" gap="25000" />
                    <frameEntry msg="5" gap="25000" />
                </frame>
        </major_frame>
        <major_frame loop="1">
                <frame>
                    <frameEntry msg="2" gap="10000" />
                    <frameEntry msg="6" gap="10000" />
                </frame>
        </major_frame>
        <major_frame loop="25">
                <frame>
                    <frameEntry msg="4" gap="20000" />
                    <frameEntry msg="8" gap="20000" />
                </frame>
        </major_frame>
        <major_frame loop="1">
                <frame>
                    <frameEntry msg="3" gap="10000" />
                    <frameEntry msg="7" gap="10000" />
                </frame>
        </major_frame>
    </stages>
</bus_controller>
```

Appendix F

CASE STUDY REMOTE TERMINAL 1 XML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<remote_terminal xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
     xsi:noNamespaceSchemaLocation="rt_schema.xsd" id="1">
     <transmit subAddr="1">
          <data fileName="scenarios/1760powerup/storeDescription.xml
              " />
     </transmit>
     <transmit subAddr="2">
          <data fileName="scenarios/1760powerup/storeMonitor.xml" />
     </transmit>
     <receive subAddr="3" />
     <receive subAddr="4" />
</remote_terminal>
```

Appendix G

CASE STUDY REMOTE TERMINAL 2 XML FILE

```
<?xml version="1.0" encoding="UTF-8"?>
<remote_terminal xmlns:xsi="http://www.w3.org/2001/XMLSchema-
    instance"
     xsi:noNamespaceSchemaLocation="rt_schema.xsd" id="2">
     <transmit subAddr="1">
          <data fileName="scenarios/1760powerup/storeDescription.xml
              " />
     </transmit>
     <transmit subAddr="2">
          <data fileName="scenarios/1760powerup/storeMonitor.xml" />
     </transmit>
     <receive subAddr="3" />
     <receive subAddr="4" />
</remote_terminal>
```