RECONFIGURABLE COMPUTING: NETWORK INTERFACE CONTROLLER AREA NETWORK (CAN)

A dissertation submitted to
The University of Manchester Institute of Science and Technology
for the degree of MSc.

By:

ABS Mohd Saman

Under the supervision of:

Dr Martyn Edwards

Computation Department

2002

ABSTRACT

In current embedded computer system development, the methodologies have experienced significant changes due to the advancement in reconfigurable computing technologies. The availability of large capacity programmable logic devices such as field programmable grid arrays (FPGA) and high-level hardware synthesis tools allows embedded system designers to explore various hardware/software partitioning options in order to obtain the most optimum solution.

A type of hardware synthesis tool that is gaining significant footing in the industry is Handel-C, a programming language based on the syntax of C but able to produce gate-level information that can be placed and routed on to an FPGA.

Controller Area Network (CAN) is an example of embedded system application widely used in modern automobiles and gaining popularity in manufacturing environments where high-speed and robust networking is needed. CAN was designed on a very simple yet effective protocol where messages are identified by their own unique identifiers. Message collisions are handled through a non-destructive arbitration process, eliminating message re-transmission and unnecessary network overloading.

A project to design and implement of a version of CAN is presented in this dissertation. The project was performed based on hardware/software co-design methodology with the utilisation of the above-mentioned reconfigurable computing technologies: FPGA and Handel-C. This dissertation describes the concepts of hardware/software co-design and reconfigurable computing; the details of CAN protocol, the fundamentals of Handel-C, design ideas considered and the actual implementation of the system.

CONTENTS

1 Introduction 1

1.1 Embedded Systems 1
1.1.1 Embedded System Design 2
1.1.2 Hardware/Software Co-design 3
1.2 The Project 4
1.2.1 Outcomes/Deliverables 5
1.3 Overview of the Dissertation 6

2 Background Discussion 8

2.1 Hardware/Software Co-design 8
2.1.1 FPGA 10
2.2 Controller Area Network 11
2.2.1 Local Area Network 12
2.2.2 OSI Reference Model 12
2.3 The CAN Protocols 13
2.4 CAN bus 14
2.4.1 CAN Physical Layer 15
2.5 Related Research 16
2.5.1 Implementations 17
2.5.2 Academic Research 17
2.6 Summary 18

3 Detailed Requirements 19

3.1 The CAN Protocol 19
3.1.1 Frame Types 19
3.1.2 Data Frame 20
3.1.3 Remote Frame 22
3.1.4 Error Frame 22
3.1.5 Overload Frame 23
3.2 Arbitration 23
3.3 Error handling 25
3.4 Handel-C 26
3.4.1 Advantages of using Handel-C 27
3.5 Handel-C Feature Highlights 27
3.5.1 Data Widths 27
3.5.2 Bit Selection 29
3.6 Physical Resources 29
3.7 Celoxica DK1 30
3.7.1 Build Options 32
3.7.2 Debugging 33
3.7.3 Targeting hardware via EDIF 34
3.8 The RC1000-PP Card 35
3.8.1 Configuring the FPGA 35
3.8.2 Software-hardware communications 36
3.9 Summary 38

4 Design of the CAN Network 39

4.1 Approach 39

4.2 High-level Design Ideas 41
4.3 CAN Bus Emulation 42
4.3.1 Message Passing 42
4.3.2 Function Emulation 43
4.4 Arbitration Handling 44
4.5 Hardware/Software Partitioning 46
4.5.1 Partitioning strategy 46
4.5.2 Prototyping 47
4.5.3 Software Simulation 47
4.5.4 Porting C to Handel-C 48
4.5.5 Hardware Simulation 49
4.6 Summary 50

5 Implementation 51

5.1 The System 51
5.1.1 Walkthrough 52
5.2 The Host Program 55
5.2.1 Functions 57
5.3 The FPGA Program 59
5.3.1 Writing to the Bus 59
5.3.2 Reading from the Bus 62
5.4 Host – FPGA communications 64
5.4.1 Host-FPGA Message Transmission 68
5.4.2 Data Buffers for Host-FPGA communication 69
5.5 CRC Calculation 70
5.6 Summary 71

6 Evaluation/Testing 72

6.1 Internal Evaluation 72
6.2 External Evaluation 73
6.3 Meta-level Evaluation 74

7 Conclusion & Further Work 77

7.1 Conclusion 77
7.2 Further Work 78
7.2.1 Parallelising 78
7.2.2 Physical Bus 79
7.2.3 Graphical User Interface 80
7.2.4 CRC Calculation 80
7.3 Summary 81

LIST OF FIGURES

Figure 2.1 – Conventional Embedded System Design Process 9
Figure 2.2 – United Design Environment of Hardware/Software Co-design 10
Figure 2.3 – OSI Reference Model 13
Figure 2.4 – OSI Layers in CAN 14
Figure 2.5 – Physical and Electrical Organisation of a CAN Bus 15
Figure 3.1 – Data Frame of CAN 2.0 A (Standard) 20
Figure 3.2 – Data Frame of CAN 2.0B (Extended) 21
Figure 3.3 – A Remote Frame of CAN 2.0A (Standard) 22
Figure 3.4 – Error Frame 22
Figure 3.5 – An Example of CAN Arbitration Process 23
Figure 3.6 – Arbitration Flow Chart 24
Figure 3.7 – DK1 Main Display Showing Its Four Main Components 31
Figure 3.8 – Host-FPGA DMA of On-board Memory 37
Figure 3.9 – An Example of Host-FPGA Communication Process 38
Figure 4.1 – Development Plan (Based on HW/SW Co-Design Model) 40
Figure 4.2 – The Proposed Network Logical Configuration 41
Figure 4.3 – CAN Bus Emulation by Message/Token Passing 43
Figure 4.4 – CAN Bus Function Emulation 44
Figure 4.5 – The Write & Read Cycle 45
Figure 4.6 – Hardware/Software Partitioning Strategy 48
Figure 5.1 – Basic Setup of the Host and FPGA Programs 51
Figure 5.2 – Host Program's Menu 53
Figure 5.3 – Host sends a message to Node 0 53
Figure 5.4 – Host receives messages from Node 1 & 2 53
Figure 5.5 – Getting Status Reports from FPGA 54
Figure 5.6 – Host's Main Flowchart 56
Figure 5.7 – FPGA's Main Flowchart 56
Figure 5.8 – Flowchart for write_frame() function 63
Figure 5.9 – Flowchart for read_frame() function 65
Figure 5.10 – Sending Message from Host to FPGA 67
Figure 5.11 – Message Buffer Format 69
Figure 5.12 – Type 2 Buffer Contents (CAN Nodes Status) 69
Figure 5.13 – CRC Calculation Algorithm 70

LIST OF TABLES

Table 2.1 – Physical Characteristics of CAN Bus 16
Table 3.2 – Host-FPGA Communication Functions 37
Table 5.2 – Types of Messages from Host to FPGA 57
Table 5.3 – Host-FPGA Interaction 58
Table 5.4 – Tasks of controller_write() function 60
Table 5.5 – Tasks of controller_read() function 64

1 Introduction

To most people a computer is equipment that we regularly use in the office, school or home to perform various tasks such as word processing, accounting and desktop publishing. Technically the type of computer that we are familiar with is called a general purposed computer. It is designed to perform various tasks from serious work such as database management to entertainment such as watching movies.

Another type of computer that we use everyday but seldom see it as a computer is called an embedded system. Embedded systems can be found in modern domestic appliances such as washing machines, dishwashers and microwave ovens. Embedded systems can also be found in cars (e.g. auto-cruising and anti-lock braking systems), digital cameras, digital televisions, CD players, mobile telephones and a lot of others. Since the computer is embedded into a larger device, people seldom think of it a computer when they use it. In fact we use more embedded computer systems everyday compared to general purpose computers such as the personal computer (PC).

1.1 Embedded Systems

An *embedded system* is a combination of computer hardware and software designed to perform a specific function. It is a part of a larger system that may not be a computer. A general purposed computer like the PC is built on a general purposed hardware subsystem. Different software subsystems can be loaded on top of the hardware subsystem to perform different tasks. Unlike a PC, which can be used for a

variety of tasks, an embedded system performs a specific and fixed function. Its hardware subsystem is built from the outset to perform this function in the most efficient manner. Its software subsystem is written to complement the hardware. Because of this, embedded systems are usually very small and perform their intended function very efficiently [1].

Hardware is used mainly because of its performance. A system built on hardware is thousands of times faster than an equivalent software system. A typical software system contains several layers of hardware and software, thus adding huge amount of overhead to the overall performance of the system. However, hardware is less flexible. On the other hand, software is more flexible and easier to update. Thus, the software subsystem is designed to provide features to the embedded system.

1.1.1 Embedded System Design

Traditionally, developing an embedded system was done by writing a piece of software to suit a particular hardware architecture. The hardware is usually based on a certain type of microprocessor. A variety of microprocessors and microcontrollers with different features and strengths have been developed and produced by integrated circuit makers for different areas of application. There are microprocessors of varying data sizes (i.e. 8-bit, 16-bit, 32-bit etc.) developed for general purpose, and there are also co-processors developed for specific purpose such as image processing (e.g. digital signal processors), mobile communications (e.g. Motorola MX1 processor) and internet appliances (e.g. Philips TriMedia processor). Certain hardware is more suitable for small appliances while another is specifically designed for use in a harsh environment. Because the hardware sub-systems are pre-developed, the software sub-systems can only be developed after the hardware has been identified. [2,3]

Design of embedded systems can be subject to many different types of constraints, including timing, size, weight, power consumption, reliability, and cost. Conventional methods for designing embedded systems require engineers to specify and design hardware and software separately. A specification, often incomplete and written

in a non-formal language, is developed and sent to the hardware and software engineers. The hardware-software partition is decided *a priori* and is adhered to as much as is possible, because any changes in this partition may necessitate extensive redesign. Designers often strive to make everything fit in software, and off-load only some parts of the design to hardware to meet timing constraints [7].

1.1.2 Hardware/Software Co-design

There are many different approaches of trying to solve the problem of embedded system design. Each has its own strengths and weaknesses. Some are more suitable to certain types of applications compared to others. With the advent of programmable hardware such as *Application Specific Integrated Circuits* (ASIC), the hardware can be designed and built in tandem with the software development — a methodology known as *hardware/software co-design*. In this method, the system's functions are partitioned into hardware and software sub-systems, developed separately, optimised, and finally integrated. A more detailed discussion about this methodology is presented in Chapter 2.

Reconfigurable devices such as *Field Programmable Grid Arrays* (FPGA) were often used for prototyping the ASIC designs [5]. Today, FPGAs are powerful and cheap enough to be used as the target hardware — giving birth to the *Reconfigurable Computing System Development* methodology. With this method, the hardware sub-system invariably contains reconfigurable computing resources (usually FPGA) together with conventional processor. The processor takes care of the general-purpose computation while the reconfigurable hardware takes care of specific applications. The software sub-system is normally developed on a personal computer (PC) connected to the hardware.

A major advantage of a reconfigurable computing system is that the hardware sub-system can be reconfigured to suite changes in the application requirements. The execution speed of dedicated hardware is retained but there is a great degree of functional flexibility. The logic within the FPGA can be changed if or when it is

necessary. For example, hardware bug fixes and upgrades can be administered as easily as in software. Obviously, during system development changes can be as often as needed in order to explore various configurations and features, with the objective of producing the most optimum solution possible.

1.2 The Project

In this project, the process of designing and implementing an embedded system using reconfigurable computing technology was explored. The embedded system application implemented was a network interface using the Controller Area Network (CAN) bus protocols. The system consisted of a hardware sub-system (on an FPGA) and software sub-system (on a Personal Computer). Embedded system design methodology and reconfigurable computing techniques were applied throughout the project.

CAN is a serial communications protocols which efficiently supports distributed real-time control. It is commonly employed as a Local Area Network (LAN) to interconnect electronic devices in automobiles, thus sometimes referred to as *Car Area Network*. However, due to its simplicity and flexibility, it is receiving widespread use in a wide variety of embedded applications like industrial control where high-speed communication is required [23]. The fundamentals of CAN are discussed in Chapter 2 and further treated in detail in Chapter 3.

The aim of the project was to implement the functionality of a Controller Area Network (CAN) bus using hardware and software. In order to achieve this aim, the following functional objectives have been defined:

- To demonstrate the operation of CAN as a network interface.
- To demonstrate the operation of three CAN controller nodes communicating with each other using CAN protocol.

- To demonstrate the control of the network operation using a personal computer (PC) interfaced to the hardware (FPGA).

In achieving those objectives, the following non-functional objectives were also defined:

- To deliver the implementation in a system of mixed hardware and software.
- To design and develop the system using the practical methods and techniques normally employed in a typical embedded system development environment.
- To design and implement the system using the concepts of *Reconfigurable Computing*.

In this project the hardware subsystem was built on Field Programmable Grid Array (FPGA) while the software subsystem was written on a PC. The software subsystem was written in C/C++ using Microsoft Visual C++. The hardware was developed using Handel-C Programming Language — a hardware design language that is gaining significant footing the hardware/software co-design world.

Handel-C is a high-level language based on ISO/ANSI C for the implementation of algorithms in hardware. It includes extensions to C that provide features for describing the behaviour of embedded systems in hardware [4]. Basic features of Handel-C are discussed in Chapter 3.

1.2.1 Outcomes/Deliverables

The desired deliverable was a mixed hardware/software implementation of a CAN bus network interface. Several CAN "devices" interconnected via a CAN bus were to be built into an FPGA. The devices were to communicate with each other via the bus and controlled by programs running on a PC connected to the FPGA. Essentially, it was aimed to be a network that consists of:

- A CAN bus – emulating two pieces of wire that normally required in a CAN bus.
- CAN devices – at least three simulated CAN devices communicating with each other.
- Network Monitor/Controller – the PC will be used to monitor the network and for initiating data transfer from one node to another.

At the end of this project, it was desired that the author would gain significant insight into the practical aspect of embedded system development. Along the way, significant understanding of the concept of *Reconfigurable Computing* was also aimed for. Valuable knowledge in the operation of CAN as a network interface and experience gained in the utilisation of Handel-C as a major part of the system development, can be shared through this dissertation.

1.3 Overview of the Dissertation

This dissertation is divided into seven chapters. This introductory chapter gives an overview of the project. In Chapter 2, important concepts are introduced and treated in more detail in order to set the appropriate background for further discussions in subsequent chapters. These concepts include Embedded System Design Methodology, CAN fundamentals, CAN protocols and CAN bus. Chapter 3 sets the requirements of the project by discussing specific CAN concepts in more detail. It will introduce the CAN data frame format and indicate how CAN handles arbitration. Handel-C, which is an important element of the project will also be discussed here. Several important Handel-C constructs and functions will be described in detail.

Chapter 4 builds upon the background and requirements set in the preceding chapters. It introduces several design ideas and evaluates their strengths and weaknesses. It also discusses the approach taken for the design and development of the system. Chapter 5 describes the final product and also highlights how particular critical issues, for example hardware-software communications are resolved.

The system is the evaluated in Chapter 6. Evaluations are performed in relation to the objectives and requirements identified in preceding chapters. Chapter 7 concludes the dissertation by revisiting important elements of the project and by looking forward to possible extensions to the current project.

2 Background Discussion

This chapter presents the background to the area of investigation and establishing the context of the problem. Several important computing concepts will be introduced here. We will start with the basics of embedded system development, continued by the motivation behind the development of reconfigurable systems.

After establishing these important fundamentals, Controller Area Network (CAN) will be introduced. Befitting its role as the backbone of this project, the fundamentals and basic operation of CAN are covered in detail.

2.1 Hardware/Software Co-design

In hardware/software co-design methodology, hardware engineers and software engineers work on their designs in parallel. Given a list of requirements, designers consider trade-offs in how hardware and software components work together. Naturally, there is a need for good feedback and interaction between the two groups of designers. Decisions are evaluated on performance, programmability, area, power, development and manufacturing costs, reliability, maintenance etc. The ultimate aim is to exploit the synergy between hardware and software. [5]

A critical issue in most hardware-software co-design is finding effective hardware and software partitioning early in the process (Figure 2.1). Early system partitioning means that designers are clearly aware of the extents of their designs. However, early system partitioning also means that optimisation can only be done at the sub-system

level. If the partitioning were later discovered not to give the optimum price/performance ratio, it still has to be used as it is.

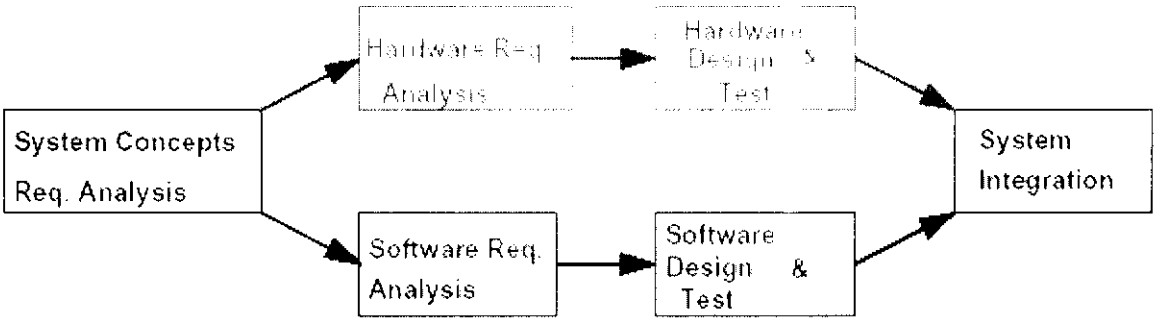

Figure 2.1 – Conventional Embedded System Design Process [5]

In the conventional design process, early partitioning results in what is known a *Model Continuity Problem* i.e. the unavailability of reconfiguration options once the partitioning has been done. Model continuity is important because many complex systems do not perform as expected in their environment. Continuity allows the validation of system level models at all levels of hardware/software implementation, thus trade-offs are easier to evaluate at several stages. [5]

Today, the availability of mature high-level and logic-level synthesis tools made it possible for various partitioning options to be simulated and evaluated. These tools allow for systematic exploration of trade-offs of hardware/software partitioning at the system level. They bridge the gap between and algorithmic specification and its implementation at the layout level. They also add a great degree of automation in hardware/software co-design. [6]

In hardware/software co-design, interaction and the need for reconfiguration during the whole of the design process is greatly emphasised. A typical co-design process flow is shown in Figure 2.2 where incremental evaluations are done at various stages of the development process [5]. Re-configuration is made possible by advancement in reconfigurable computing technologies especially FPGA and hardware synthesis tools.

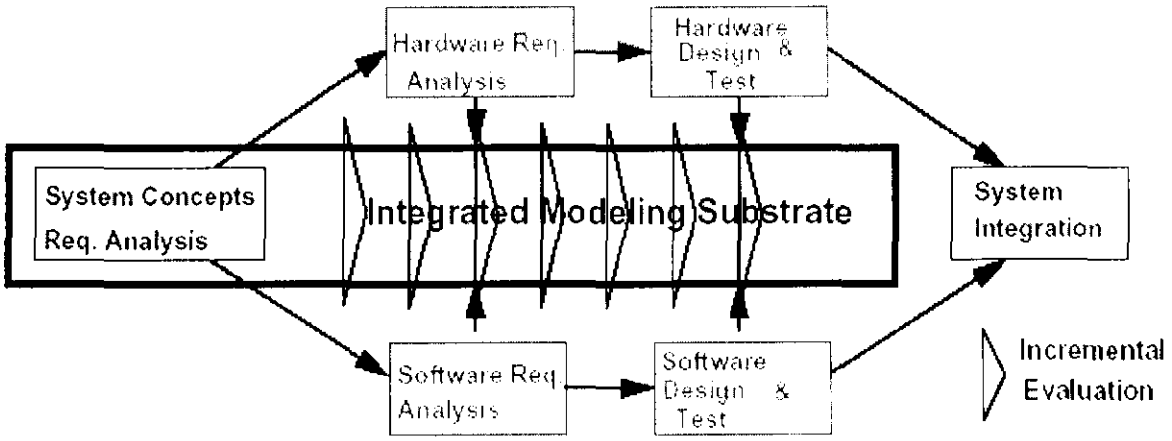

Figure 2.2 - Unified Design Environment of Hardware/Software Co-design [5]

2.1.1 FPGA

FPGA is a type of programmable device that can be configured for a wide variety of applications. Before FPGA, PLDs are generally limited to hundreds of gates, while FPGAs support thousands or even millions of gates. These gates and their interconnects are user-programmable. Some FPGAs include other logic elements such as random access memories, flip-flops and input/output buffers. By programming, other logic elements can be synthesised. Thus an FPGA can be programmed to perform a huge variety of functions.

Usually *hardware description languages* such as *VHDL* and *Verilog* are used to describe the logic to be synthesised in an FPGA. A hardware description language can be used to describe the hardware at different levels of abstraction i.e. gate level, register transfer level and behavioural level [7]. Computer Aided Design (CAD) vendors typically include various other tools such as simulator, performance analyser and system verifier. [26]

Alternatively, the hardware can be described algorithmically — like software programming — using a conventional programming language variance or subset such as Handel-C. This alternative is becoming more attractive because current embedded systems are becoming more complex and require complex algorithmic solutions

equivalent to those employed in large software systems. The Handel-C programming language is discussed in Chapter 3.

2.2 Controller Area Network

Controller Area Network (CAN) is a serial communications protocols which efficiently supports distributed real-time control. It is a type of network that was designed to efficiently support distributed real-time control with a very high level of security [8]. It is commonly employed as a Local Area Network (LAN) to interconnect electronic devices in automobiles. Microchips manufacturers usually categorised their CAN-related products under *In-Vehicle Networking* [12, 13] because CAN was developed in the automotive industry. However, its domain of application ranges from high speed networks to low cost multiplex wiring in vehicles and manufacturing environment.

In automotive electronics, engine control units, sensors, anti-skid systems and others are connected using CAN because it is physically easier to install compared to conventional point-to-point wiring. It also requires a minimum amount of cables and connectors, thus weighs less. Effectively it is more cost-effective compared to the normal wiring harness. Another important reason of using CAN in vehicles is to enable any station to communicate with any other without putting too great a load on the controller computer [23].

Fundamentally, CAN is a type of Local Area Network (LAN). It is built on a collision-detection broadcast bus similar to *Ethernet*, a very popular type of LAN [9]. However, in Ethernet collision-detection forces conflicting message senders to stop and resubmit their messages after a random interval. In CAN collision-detection signals the message senders to go into a non-destructive arbitration process. This will be discussed further in Section 3.2.

2.2.1 Local Area Network

A Local Area Network can be defined as a network of computers and other devices in a limited geographical area such as in a building or within a campus area. A common transmission medium is shared by all the participating devices, through which they communicate with each other. They also share resources such as storage and printing devices.

The communication and resource sharing is made possible because a network is always built using well-defined hardware and software specifications. These standard specifications allow the network to be built systematically and operated smoothly.

2.2.2 OSI Reference Model

Modern computer networks are designed in a highly structured way. To reduce their design complexity, most networks are organized as a series of layers, each one built upon its predecessor. This structure is known as the OSI Reference Model (Figure 2.3), which is divided into seven layers which can be described as follows [10]:

Layer 7: **Application** : Provides services that meet the communication requirements of specific applications, often defining the interface to a service.

Layer 6: **Presentation** : Transmits data in a network representation that is independent of the representations used in individual computers.

Layer 5: **Session** : Handles problems which are not communication issues such as detection of failure and automatic recovering

Layer 4: **Transport** : Provides end to end communication control

Layer 3: **Network** : Routes the information in the network

Layer 2: **Data Link** : Provides error control between adjacent nodes

Layer 1: **Physical** : Connects the entity to the transmission media

Layering brings substantial benefits in simplifying and generalising the software interfaces for access to the communication services of a network.

2.3 The CAN Protocols

The fundamental design of CAN has been mapped to the Data Link and Physical layers of the ISO/OSI Reference Model. The Data Link layer of CAN is further subdivided into two sublayers: Logical Link Control (LLC) and Medium Access Control (MAC) sublayers (Figure 2.4). The scope of the LLC sublayer contains the following functions:

- to provide services for data transfer and for remote data request

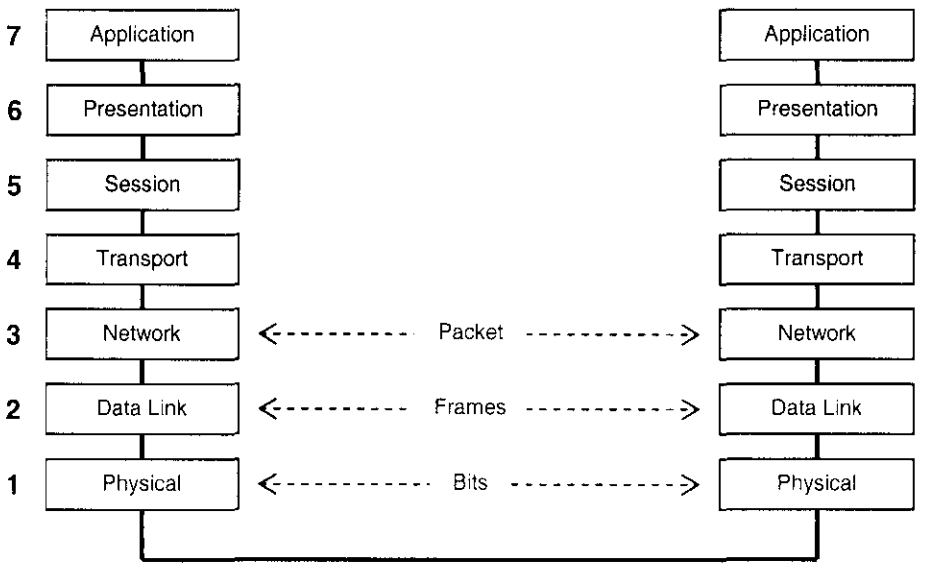

Figure 2.3 - OSI Reference Model

- to decide which messages received by the LLC sublayer are actually accepted
- to provide means for recovery management and overload notifications.

CAN employs content-oriented addressing scheme. Communication is addressed by message identifiers instead of station identifiers as in normal LAN. Each message has an identifier that is unique throughout the network. It defines the priority and the

content of the message. When a station transmits a message, all other stations in the network become receivers. The LLC sublayer in each station will perform an acceptance test to determine whether the data received are relevant for that station. If the data is of significance for the station concerned, it is processed, otherwise it is ignored.

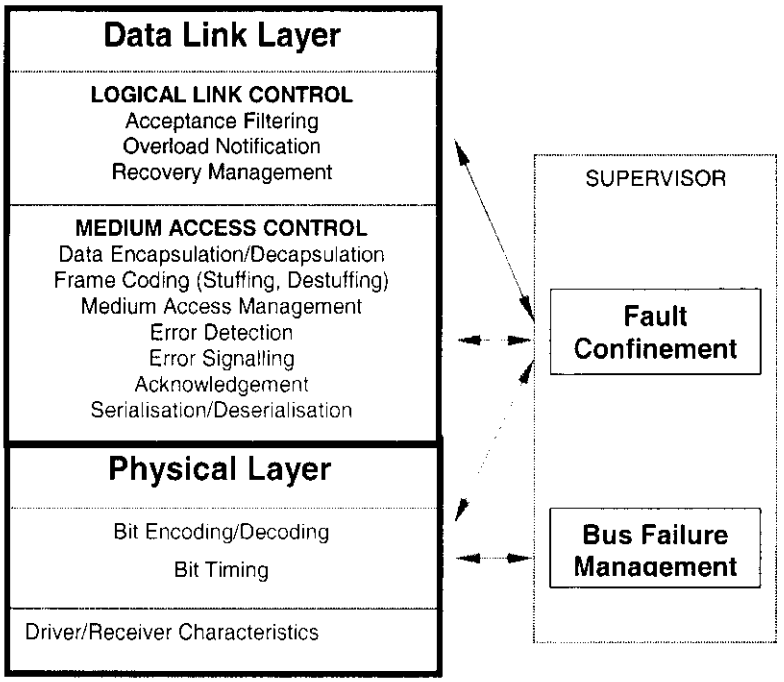

Figure 2.4 - OSI layers in CAN

In other words, the sender of a message send a broadcast throughout the network and the receivers listen to the message and decide whether to act on it or not. No physical destination is required. Since the data transmission protocol does not require physical destination addresses for individual stations, the system has some degree of configuration flexibility. Nodes can be added to or removed from the network without bringing it down as long as the said stations are purely receivers.

2.4 CAN bus

The bus in a CAN network is a serial communication link onto which a number of nodes may be connected. The maximum number of nodes is only limited by delay

times and/or electrical loads on the bus line [8]. The bus consists of a single channel that carries bit values. Physical implementation of the bus is fixed by CAN specification, thus there can be several implementations (see Table 2.1).

Bitstreams on the bus are coded according to the Non-Return to Zero (NRZ) method with bit-stuffing. The two logical bit values on the bus is known as *dominant* and *recessive*. When there is a simultaneous transmission of dominant and recessive

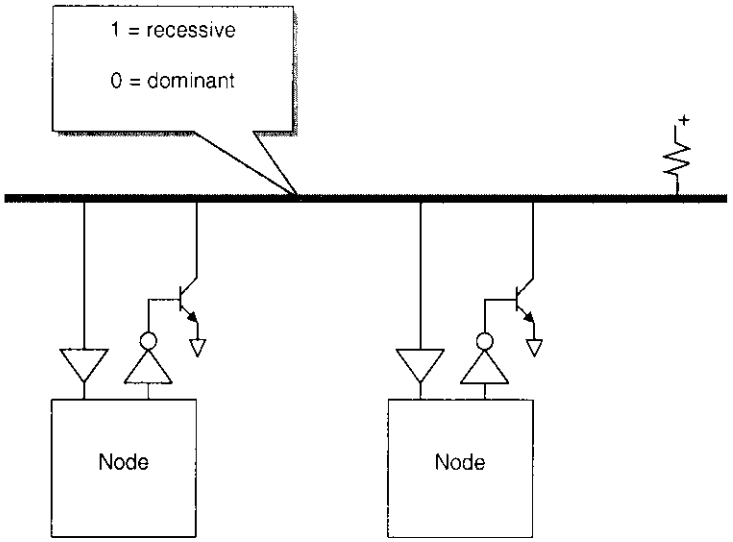

Figure 2.5 - Physical and electrical organisation of a CAN bus – wired-AND implementation [7]

bits, the bus will read as dominant. In the popular wired-AND implementation of the bus (Figure 2.5) [7], the dominant value is represented by a *logical 0*, while recessive by a *logical 1* [8].

2.4.1 CAN Physical Layer

Physically, a CAN bus is essentially a cable consisting two pieces of wire which are twisted over each other along their length. This type of cable is usually known as *twisted pair* — the most common type of cable used in normal LAN. Usually, CAN nodes are connected to the bus in a *wired-AND* fashion — if one node is writing a dominant bit (LOW) to the bus, then the whole bus is in dominant state, regardless of

the number of other nodes transmitting recessive (LOW) bits. Basic characteristics of a CAN bus is listed in Table 2.1 [22].

Characteristic	Value
Standards	▪ ISO 11989 – Two-wire balanced signalling scheme ▪ ISO 11519 – Low-speed two-wire balanced signalling scheme
Maximum bus speed	▪ 1 Mbit/s
Maximum cable length	▪ 40 meters at 1 Mbit/s ▪ 100 meters at 500 kbit/s ▪ 200 meters at 250 kbit/s ▪ 500 meters at 125 kbit/s ▪ 6 kilometers at 10 kbit/s
Cable type	▪ 108 to 132 Ohms ▪ Twisted pair
Connectors	▪ 9-pin DSUB proposed by CiA ▪ 5-pin Mini-C and/or Micro-C, used by DeviceNet and SDS ▪ 6-pin Deutch connector, proposed by CANHUG for mobile hydraulics

Table 2.1 - Physical Characteristics of CAN bus

2.5 Related Research

Research in in-vehicle networking has resulted in many standards developed by various manufacturers and organisations. CAN is one of the few that are more popular than the rest. Large microchip manufacturers such as Intel, Philips and Fujitsu have produced several CAN implementations of their own. And further research into CAN has resulted in several extensions such as CANopen, DeviceNet and CAN Kingdom.

CANopen is a CAN-based higher layer protocol originally developed for industrial control systems. The family of specifications includes also different device profiles as well as frameworks for specific applications. DeviceNet is also a CAN-based higher layer protocol developed based on an object-oriented communications model

[22]. CAN Kingdom is also another CAN-based higher layer protocol but was designed based on the concept of customisable network. In conventional network concept, devices connected to the network must be tailor-made to the network. Thus the system must conform to the network. In CAN Kingdom, the network will be tailor-made to suit the needs of the system. The system designer can create systems using virtually any type of bus management and topology – possibly making the system very flexible to the extent of making it very restrictive [11]

2.5.1 Implementations

Robert Bosch GmbH developed the CAN controller in the early 1980s and worked with Intel on the first implementation. The first implementation, 82526 controller, was based on CAN version 1.2 while the latest controller released in 1993, the 82527, supports CAN version 2.0B [12]. Intel's programming model of CAN implementation is known as *Full CAN* while those implemented by Philips is known as *Basic CAN*. Most CAN controllers allow for both programming models to be used -- and they are compatible with each other [22].

A few of other commercial organisations actively involved with CAN are Philips, Acutest, *I+ME actia, and Hitex.* Philips produces several versions of on-chip CAN controllers based on the popular 80C51 microcontroller family [13] and a few standalone CAN controllers as well [14]. Smaller companies like Acutest and I+ME actia make use of chips produced by larger companies, such as Intel and Philips, to conduct research on the application of CAN, particularly in the areas of in-vehicle networking and manufacturing automation.

2.5.2 Academic Research

Academic research in CAN is usually linked to *real-time systems* such as the analysis done by Tindell et al [9] in which an idealised scheduling analysis for CAN was derived. A study on two CAN chips (Intel 82527 and Philips 82C200) were also done using the scheduling theory derived. Although CAN was originally designed for

in-vehicle network, today, CAN is used in applications other than in automotive electronics. Some studies of CAN applications in *machine control systems* were done by Fredriksson [15] and Zuberi & Shin [16]. Studies on CAN systems performance have been done by Rauchaupt [17] and Upender & Dean [18] among others.

A work similar to this project was done by Lemus et al [19] — the implementation of CAN controllers as communication nodes in a distributed system. The controller was modelled using a hardware description language i.e. VHDL as opposed to this project, where the hardware will be programmed by a programming language variation, i.e. Handel-C. Twenty CAN controllers were connected the bus and their operational behaviour were studied. It was noted that the controllers used a global clock because bit synchronisation was not implemented.

At the "physical layer", work on defining a single-wire CAN bus is in progress but the standard has not been established yet [22]. Also, there are several *Higher Layer Protocols* already developed and their practical application being studied [20]. Other current research involved techniques for combining CAN with Bluetooth. Fredriksson of KVASER discussed the possibilities and concerns in this area [21].

2.6 Summary

In this chapter, we have looked at the concepts of hardware/software co-design and the fundamentals of CAN. We have discussed how the OSI layers in CAN relate to the lowest two layers of the OSI Reference Model. We have also seen that physically a CAN bus is very simple and easy to implement. Details of CAN such as its data format, arbitration and error detection are discussed in next chapter.

3 Detailed Requirements

In order to identify the requirements of the project, we need to investigate the protocol of CAN. This chapter describes the details of CAN protocol and how CAN handles arbitration and errors. As the hardware synthesis was done using Handel-C, this chapter also introduces the basic of the Handel-C programming language and specific features which make it suitable for the intended purpose. A major part of the information in this chapter is obtained from technical specifications of CAN and Handel-C.

3.1 The CAN Protocol

There are two versions of CAN, each of which is based of CAN specification 2.0A and 2.0B. The two versions differ in the size of their identifiers. CAN 2.0A has a standard 11 bit identifier while CAN 2.0B has an extended frame containing a 29 bit identifier. CAN controllers from both versions can co-exist in the same network as long as the 2.0B type controllers send standard frames only [22].

3.1.1 Frame Types

Message transfer is manifested and controlled by four different frame types:

- **Data Frame** – carries data from a transmitter to the receiver
- **Remote Frame** – transmitted by the bus unit to request the transmission of the Data Frame with the same identifier

- **Error Frame** – transmitted by any unit on detecting a bus error
- **Overload Frame** – provides an extra delay between the preceding and succeeding Data Frames or Remote Frames.

3.1.2 Data Frame

A standard CAN data frame is shown in Figure 3.1:

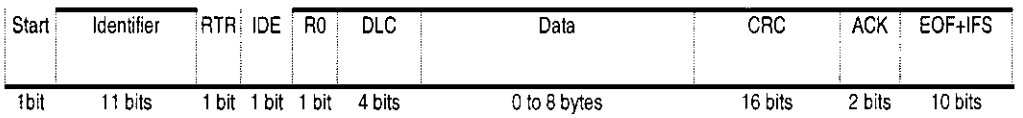

Figure 3.1 - Data frame of CAN 2.0A (Standard) [1]

The above frame consists of the following fields [8,23,24]:

- **Start Bit** (1 bit) – always LOW. Falling edge of signal from idle state (HIGH) to the Start Bit (LOW) is used for synchronisation.
- **Identifier** (11 bits) – logical *identity* and priority of the message. The smaller the value, the higher the priority – 0000 0000 000 has the highest priority while 1111 1111 111 has the lowest.
- **RTR** (1 bit) – Remote Transmission Request, set to LOW. This bit is set to HIGH in *Remoteframe*.
- **Control Field** (6 bits) – contains IDE, R0 and DLC:
  - IDE (1 bit) – Identifier Extension. Set to LOW to indicate Standard CAN Data frame.
  - R0 (1 bit) – reserved.
  - DLC (4 bits) – Data Length Code. Indicates the length of data field.
- **Data** (0 to 8 bytes) – contains the data of the message.

- **CRC** (16 bits) – Cyclic Redundancy Check. Contains the checksum of the data bits. Used for error detection.
- **ACK** (2 bits) — ACKnowledge. The first bit is the *slot* bit, which is transmitted HIGH but subsequently over-written by dominant bits from receiver nodes. The second bit is the delimiter (high).
- **EOF** (7 bits) — End of Frame. All HIGH (recessive).
- **IFS** (7 bits) — Inter Frame Space. All HIGH (recessive).

An extended CAN data frame (Figure 1.5) contains the all the fields for the standard CAN with the following differences/additions:

- **SRR** (1 bit) — Substitute Remote Request. Replaces the RTR bit in standard CAN (relocated after the identifier field). Always HIGH, thus an extended CAN frame always has a lower priority than a standard CAN frame during arbitration.
- **IDE** (1 bit) — Identifier Extension. Always HIGH to indicate extended identifier follows.
- **Identifier** (18 bits instead of 11 bits)
- **Control field** (6 bits) — now contains an additional reserved bit (r1) which replaces IDE.

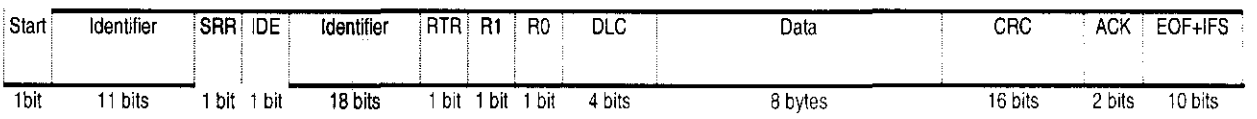

Figure 3.2 - Data frame of CAN 2.0B (Extended) [23]

3.1.3 Remote Frame

If a node wants to request certain message from another node, it will send a *remote frame* (Figure 3.3). It is identical to a *data frame* except for the following two characteristics [22]:

- The RTR bit is set to HIGH (recessive)
- There is no *Data Field.*

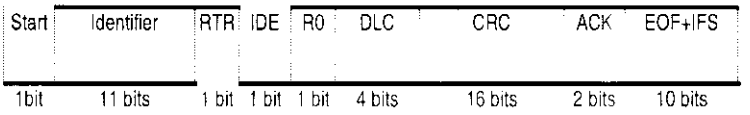

Figure 3.3 - A Remote Frame of CAN 2.0A (Standard) [23]

A receiving node that responses to the request will send out a data frame with an identifier identical to the remote frame it received. Most CAN controllers can be programmed to either automatically respond to a remote frame, or to notify the local *Central Processing Unit* (CPU) [22].

3.1.4 Error Frame

When a node detects a fault, it will send out an *Error Frame*. An error frame is a special data that violates the framing rules of CAN messaging [22]. Thus, when a node sent out an error frame, other nodes will se it as error and send out their own error frames too. If this situation occurred during transmission, the transmitter will try to re-transmit the message.

Figure 3. 4 - Error Frame

The format of an error frame is shown in Figure 3. 4. It consists of an *Error Flag* (six bits of the same value, thus violating bit-stuffing rule) and an *Error Delimiter* (eight recessive bits). The error delimiter provides enough delay for other nodes their error frames upon detecting the current one [22]. Error handling is discussed further in Section 3.3.

3.1.5 Overload Frame

An *Overload Frame* is identical to an *error frame* except that it is transmitted by node that becomes too busy. It is seldom used because today's CAN controllers are clever enough to avoid this kind of situation.

3.2 Arbitration

CAN protocol is based on CSMA/CD (Carrier Sense Multiple Access/Collision Detection) with added feature called *Arbitration on Message Priority* (CSMA/CD+AMP). A CAN node checks if the bus is busy (Carrier Sense) before sending a message. If the bus is free, several nodes could be sending at the same time

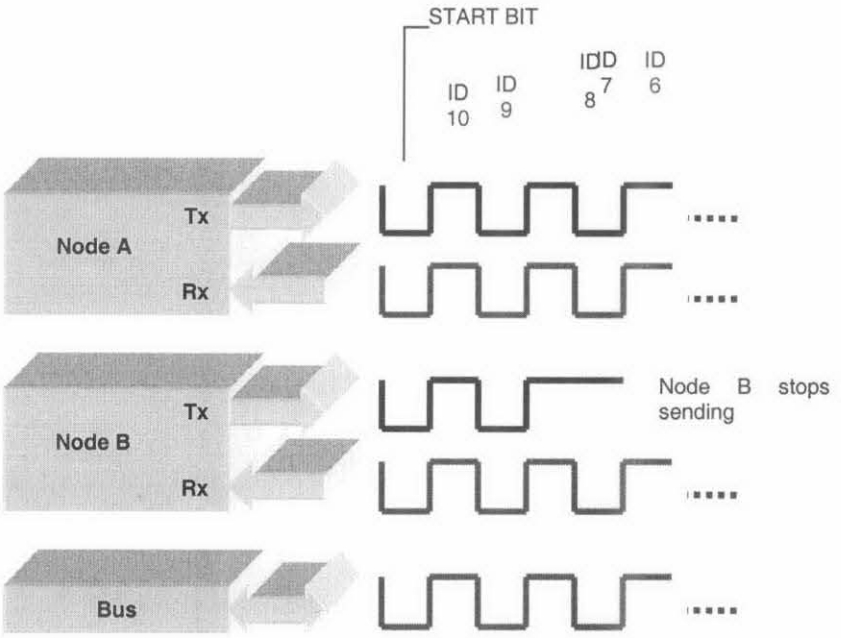

Figure 3.5 – An Example of CAN Arbitration Process [23]

(Multiple Access). Each transmitting node also checks if other nodes are also transmitting by detecting for collision. However, in Ethernet, upon detecting collision, all sending nodes will stop transmitting. They then wait for a random length of time before trying to send again — making Ethernet very sensitive to high bus loads [24].

CAN solves this problem by employing a non-destructive, bitwise arbitration [8]. The "winner" of the arbitration does not have to resend the message from beginning as happens in Ethernet. The efficiency of the arbitration depends on the physical property of the bus. When logical levels 0 and 1 are both sent to the bus, logical 0 becomes dominant and overwrite the logical 1.

If a CAN node is writing logic 0 to the bus while another is writing logic 1, the value that appears on the bus will be logic 0. After writing a bit value into the bus, each

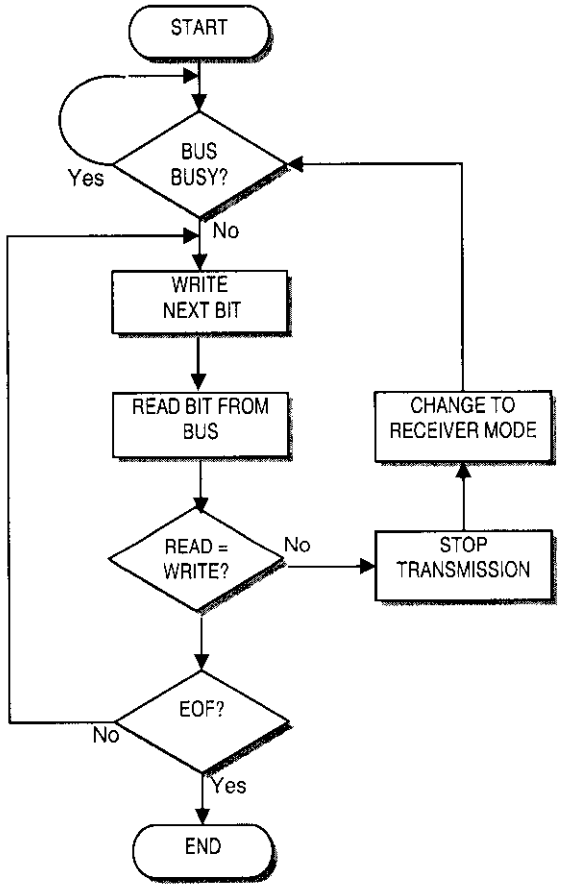

Figure 3.6 - Arbitration Flow Chart

transmitting node reads back the bit value actually registered on the bus. If a node found that the bit value it has written is different from the bit value it read back, then it will stop transmitting — it has detected a collision and has lost in the arbitration process (Figure 3.6).

The bit values that actually used in the arbitration process are those in the identifier field. Because logic 0 is more dominant compared to logic 1 (recessive), an identifier with the smallest binary value has the highest priority — always wins the arbitration process. Figure 3.5 shows an example an arbitration process of frames transmitted from two CAN nodes. A lower priority node that has lost the arbitration, switches to *receive mode*. It will then wait until the bus is idle before attempting re-transmission.

3.3 Error handling

Unlike other bus systems, the CAN protocol does not use acknowledgement messages but signals any error that occurs [25]. Its error management function, which is part of the Data Link layer, can detect the following errors:

- Bit Error – when the bit value monitored is different from the bit value written.
- Bit Stuffing Error – when 6 consecutive equal bit level is detected in frame field. Bit stuffing should have been done after each 5 consecutive equal bits.
- CRC Error – when the CRC sequence read is not identical to the one calculated.
- Form Error – when fixed-form bit field (CRC, ACK, EOF) contains one or more illegal bits.
- Acknowledgement Error – when a dominant bit is not present in the ACK field.

As mentioned in Section 3.1.4, upon error detection, an error frame is transmitted immediately. If an error is detected, the detecting node will transmit an

Error Flag and destroying the bus traffic in the process. Other nodes, upon detecting the *Error Flag* will discard the current message [8].

3.4 Handel-C

Handel-C is a programming language rather than a hardware description language. The Handel-C language syntax is based on the C programming language. Extensions have been added to support high level hardware constructs such as parallelism, concurrency, communication and scheduling. Algorithms can be expressed in Handel-C without knowing how the underlying computation engine works. This makes Handel-C a programming language rather than a hardware description language. While a conventional C generates microprocessor machine codes, Handel-C is generates hardware designs. The hardware design – at gate level -- that Handel-C produces is generated directly from the source program. The logic gates that make up the final Handel-C circuit are comparable to the machine codes in the executable file produced with conventional C. The target of the Handel-C compiler is low-level hardware.

The following is a summary of Handel-C features:

- It uses much of the syntax of conventional C.
- It has parallelism built in. By utilising parallelism, huge performance benefit can be obtained from the target hardware.
- It provides channels for communication between parallel branches of the code.
- Interface can be used to communicate with external device or component.

3.4.1 Advantages of using Handel-C

Software programs are effectively state machines. The flow of execution through the program is determined by control statements such as *if* statements, *switch* statements, *while* loops and *for* loops. Handel-C adds the par construct to implement parallelism [26]. There is also the channel statement for passing data between parallel parts of the program and for synchronising them. By writing Handel-C program to take advantage of inherent parallelism in low-level hardware, massive performance advantage can be realised.

3.5 Handel-C Feature Highlights

Handel-C parallelism is true parallelism, not the time-sliced parallelism for general purpose computers. When instructed to execute two instructions in parallel, those two instructions will be executed at exactly the same instant in time by two separate pieces of hardware.

Handel-C uses two kinds of objects: logic types and architecture types. The logic types specify variables. The architecture types specify variables that require a particular sort of hardware architecture e.g. ROMs, RAMs and channels.

3.5.1 Data Widths

A crucial difference between Handel-C and conventional C is Handel-C's ability to handle values of arbitrary width. Conventional C handles 8, 16 and 32 bit values well but cannot easily handle other widths. When targeting hardware, there is no reason to be tied to these data widths. So Handel-C has been extended to allow types of any number of bits. It is perfectly valid to use 32-bit values for all data items but a large amount of hardware is produced if none of these values exceed 8 bits. Declaring data of suitable widths allow for an optimum use of hardware.

As in conventional C, the following data types have fixed width:

```
[ signed | unsigned ] char    <— 8 bits
[ signed | unsigned ] short   <— 16 bits
[ signed | unsigned ] long    <— 32 bits
```

The syntax for declaring an integer variable of an arbitrary width is:

```
[ signed | unsigned ] int width variable_name;
```

For example, to define a 12-bit wide unsigned integer "**arbitfield**", the following declaration can be used:

```
unsigned int 12 arbitfield;
```

If the width is omitted, the variable width is classified as "undefined". During compilation, the compiler will try to infer a suitable width for the variable. However, in normal practice, each variable width is always defined the programmer.

Values of different widths can only be assigned to each other using the append operator (@) or the bit selection operator. For example, to assign the value of a 4-bit variable `x4` to an 8-bit variable `y8`:

```
unsigned int 4 x4;
unsigned int 8 y8;

y8 = 0 @ x4;
```

In this example, the left side (most significant bit side) of `x4` will be appended with extra zeros before being assigned to `y8`. In this case, four zeros will be appended to make the total number of bits eight.

3.5.2 Bit Selection

Individual bits or range of bits can be selected from a value by using the [ ] operator. Bit 0 is the least significant bit and bit n-1 is the most significant bit where n is the width of the value. For example, referring to previous example, to assign the value of four least significant bits of y8 to x4, the following statement can be used:

```
x4 = y8[3:0];
```

The above statement takes the value of bits 3, 2, 1 and 0 of y8 and assigns it to x4. Another example of bit selection assignment is as follows:

```
x4 = y8[7:4];
```

In this example, the four most significant bits of y8 is assigned to x4.

There are other bit manipulation operators in Handel-C but used less extensively in this project. Those operators are:

<<	shift left
>>	shift right
<-	Take least significant bits
\\	Drop least significant bits
@	Concatenate bits

Materials discussed here are only highlights of features used in this project. For completeness, the Handel-C Language Reference Manual should be referred to [4].

3.6 Physical Resources

These are the physical resources utilised for the design, development implementation of this project:

a. Personal Computer — Pentium class processor running Microsoft Windows 2000

b. Microsoft Visual Studio -- C++ compiler

c. Celoxica DK1 -- Handel-C compiler [28]

d. Celoxica RC1000-PP board -- FPGA programming kit/hardware [27].

3.7 Celoxica DK1

DK1 is Handel-C system development environment that have been designed with the look and feel similar to Microsoft Visual C++ (Microsoft Visual Studio). In its *debug* mode it allows programs written in Handel-C to be simulated in the environment without the need for the target hardware to be present. This feature allows the correctness of the algorithms to be tested before being applied onto the target hardware. This is discussed further in Section 4.3.2. [28]

The DK1 development environment version 2.1 used for the project is equipped with a new graphical user interface similar to Microsoft Visual C++ 6.0 (part of Visual Studio 6.0). There are four main components is the GUI (Figure):

1. Menu and tool bars
2. Workspace window
3. Editor window
4. Output window

The menu and tool bars contain drop menus (similar to standard Windows applications) and shortcut buttons to carry out major tasks (similar to Microsoft Visual Studio). Commands assigned to these buttons are, among others: compile, build and debug.

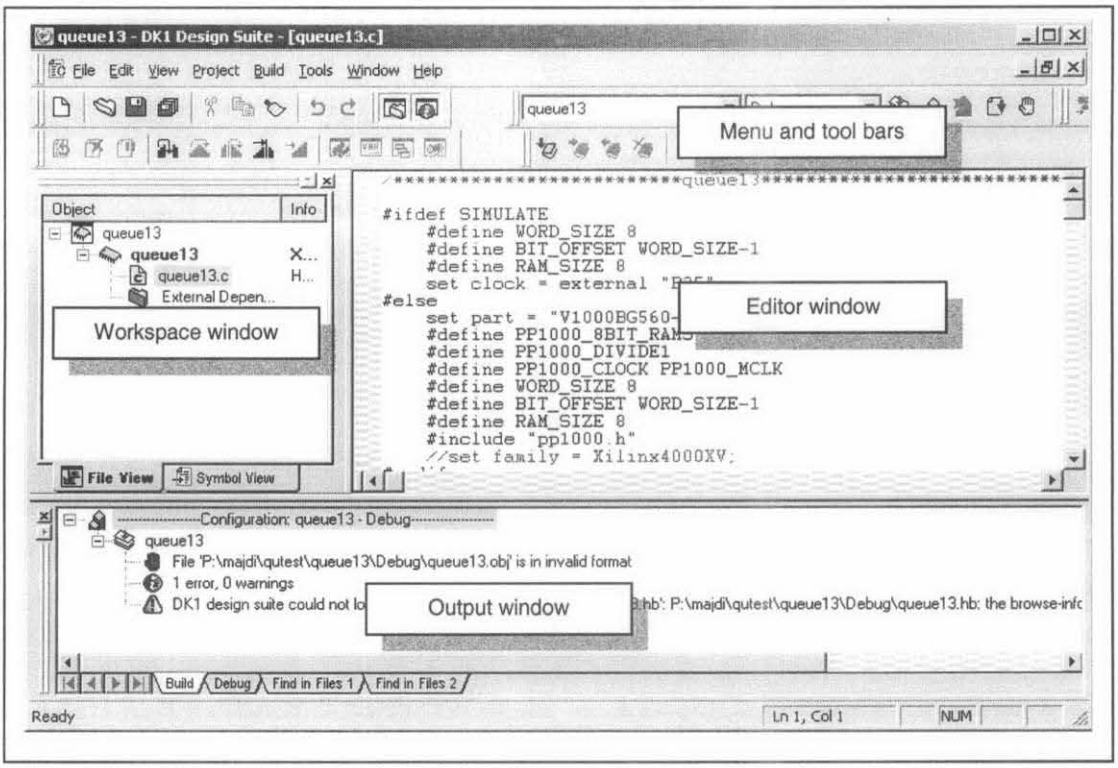

Figure 3.7 – DK1 Main Display Showing Its Four Main Components

The workspace window shows a list of source files contained in current workspace. These files are arranged in the following hierarchy:

- Workspace
  - Project 1
    - Source file 1
    - Source file 2
    - …
  - Project 2
  - …
  - Project n

At any time, only one workspace can be opened in one instance of DK1. Inside a workspace, several projects can be defined, each containing its own source files. This arrangement allows several projects (perhaps variations to the same design) to be written, compiled, built and compared.

The editor window contains the source files currently opened for editing. Several files can be opened simultaneously (each in its own window) but only one of them can be made active at any instance. The other windows are normally hidden behind the active one. The source codes are displayed in a variety of colours to distinguish their types and functions (e.g. keywords, comments, strings etc.)

When a source file is compiled or a project is built, vital information is displayed in the output window. Errors are displayed with indications and hyperlinks to their line numbers in the source file. Clicking on an error message will bring up the offending line in the editing window.

3.7.1 Build Options

DK1 allows projects to be built in several configurations: debug, release, EDIF, VHDL and generic. However only two configurations were utilized during the development of this project: Debug and EDIF. These will be discussed in the following sections. DK1 Design Suite User Manual should be referred to for completeness [ ].

Debug is the default compilation configuration. Projects built in the debug mode can be executed in the built-in simulator allowing for debugging to be done without the presence of the target hardware. This method was used at the hardware simulation stage of this project.

EDIF is one of the configurations that can be used to target a particular hardware (the other configuration is VHDL). EDIF files generated can be used for placing and routing into a targeted FPGA architecture. Obviously, this method was used at a later stage of the development when the Handel-C program was tested with an RC1000-PP card.

3.7.2 Debugging

To aid the debugging process, sample inputs for the project can be specified by the chanin keyword. For example:

```
chanin 8 Input with {infile = "data.txt"};
```

In the above example, an input channel Input (8 bit wide) was declared to read from a file named data.txt. Values in the file must be numbers only and written one number per line. They are read with the following channel operation (assuming that x has been declared as an `unsigned int 8` variable):

```
Input ! x;
```

The above statement will read one value from the data.txt file and assign it to x. If the end of the file has already been reached, a zero value will be read instead.

Outputs can be channeled either to a debug window (within the output window) or to a file using the `chanout` keyword. Declarations for output channels can be written as follows:

```
chanout Output with {infile = "output.txt"};
chanout myDebug;
```

In the first line of the above example, an output channel Output was declared to write values into a file name output.txt. In the second line, no file was specified, thus any values sent out through this channel will be displayed in the output window. Channeling values through the above channels can be done as follows:

```
Output ? x;        // output to file
myDebug ? 100;     // display in window
```

Outputs through these channels are restricted too: only one value per line can be written at any one time. Thus, if multiple output values are required, each of them must have own channel.

When executed in a simulation, various execution points in the source code are indicated with arrows of different colours i.e. current function calls (green), current execution point (yellow), combinatorial codes that will be executed on the next clock tick in other threads (white) and combinatorial codes that will be executed on the current clock cycle (grey).

Debugging can be made more effective by placing breakpoints at suitable points in the source code. Breakpoints are indicated as active (red dot), disabled (white dot with red edge) or mixed (half red, half white dot).

3.7.3 Targeting hardware via EDIF

When enough debugging has been done, the build configuration can be changed to EDIF so that the program can be tested on the actual target hardware. For the RC1000-PP card, this is a two-step process. Building the source code in the EDIF configuration is the first step. The second step is converting the EDIF files into a bitstream file that can be loaded directly into the card. This can be performed using the edifmake utility supplied with the card. [28]

The edifmake utility is a DOS batch file, therefore must be executed from a command prompt. Edifmake needs access to several files built by the compiler, thus it is normally executed in the EDIF subdirectory of the project being worked on. If the project name is cansim, a command prompt is opened under its EDIF subdirectory and the following command is entered:

```
edifmake cansim
```

A file with the name cansim.bit will created under the same subdirectory when the conversion process is completed. One method that can be used to load this bitstream file into the FPGA is by using the PP1000ConfigureFPGA() library function supplied with the RC1000-PP card. This library functions and other information about the card are described further in next section.

3.8 The RC1000-PP Card

The RC1000-PP design board is included with the DK1 design suite. The card includes a Xilinx Virtex XCV1000 FPGA with 1 million gates, 8 megabytes of RAM (in four 2 MB banks) and various expansion slots mapped to a selection of pins on the FPGA. This card can be plugged into a PCI slot on a PC and supplied with suitable drivers for Microsoft Windows. Also included are a library file and its corresponding header file, which add special commands that allow a C program running on the PC to access the card. These commands are in form of library functions defined in the header file. There is a set of functions for the C program and a corresponding set for a Handel-C program. Some of these commands (particularly those used in this project) are discussed below [27].

3.8.1 Configuring the FPGA

The first step that a host program must do is getting a handle to a RC1000PP card installed on the PC. The easiest way to do this is by opening the first card available by calling the following function call:

```
PP1000OpenFirstCard(&Handle);
```

In the above example, `Handle` is a predefined variable inside the corresponding `pp1000.h` header file supplied with the card. Then, the clock speed for the card should be set with a function call such as follows:

```
PP1000SetClockRate(Handle, PP1000_MCLK, 1e6);
```

In this example, the clock speed was set at 1MHz (1e6) with the card operating at the same speed (PP1000_MCLK). For detailed explanation of clock setting, the RC1000 Software Reference Manual should be consulted [27].

The next step is to load the bit file that has been prepared for the FPGA. This is typically done by making another function call such as follows:

```
PP1000SetClockRate(Handle, "canfpga.bit");
```

In the above example, a bit file named "canfpga.bit" was loaded into the FPGA on-board the RC1000PP card. Once loaded the program is started automatically. There are other methods of configuring the FPGA. These are covered in detail in the RC1000 Software Reference Manual [27].

3.8.2 Software-hardware communications

Data transmission between software (i.e. a host program running on the PC) and hardware (i.e. the FPGA) can be performed in three modes: 1 bit, 1 byte and direct memory access (DMA). The three types of data transfer operations are handled by the following library functions (Table 3.1):

Data Transfer	Host	FPGA
Bit-sized	`PP1000SetGPO()` – set the GPO (general purpose output) pin `PP1000ReadGPI()` – read the status of the GPI (general purpose input) pin	`PP1000ReadGPO()` – read the status of the GPO pin `PP1000SetGPI()` – set the GPI pin
Byte-sized	`PP1000WriteControl()` – send one byte of data to the FPGA `PP1000ReadStatus()` – receive one byte of data from the	`PP1000ReadControl()` – receive one byte of data from host `PP1000WriteStatus()` – send one byte of data to host

	FPGA	

DMA	`PP1000SetupDMAChannel()`
	– set up a DMA channel
	
	`PP1000RequestMemoryBank()`
	– request access to a memory bank
	
	`PP1000DDMA()`
	– execute the DMA data transfer
	
	`PP1000ReleaseMemoryBank()`
	– relinquish access to a memory bank
	
	`PP1000SetupDMAChannel()`
	-- close the DMA channel used
		`PP1000RequestMemoryBank()`
		– request for access to a memory bank
		
		`PP1000WriteBank()`
		– write to a memory bank
		
		`PP1000ReadBank()`
		– read from a memory bank
		
		`PP1000ReleaseMemoryBank()`
		– relinquish access to a memory bank

Table 3.1 - Host-FPGA Communication Functions

The third mode for data transfer, the DMA, is handled through the a set of library functions that allow direct access to the onboard memory blocks of the RC1000-PP card (see Figure 3.8).

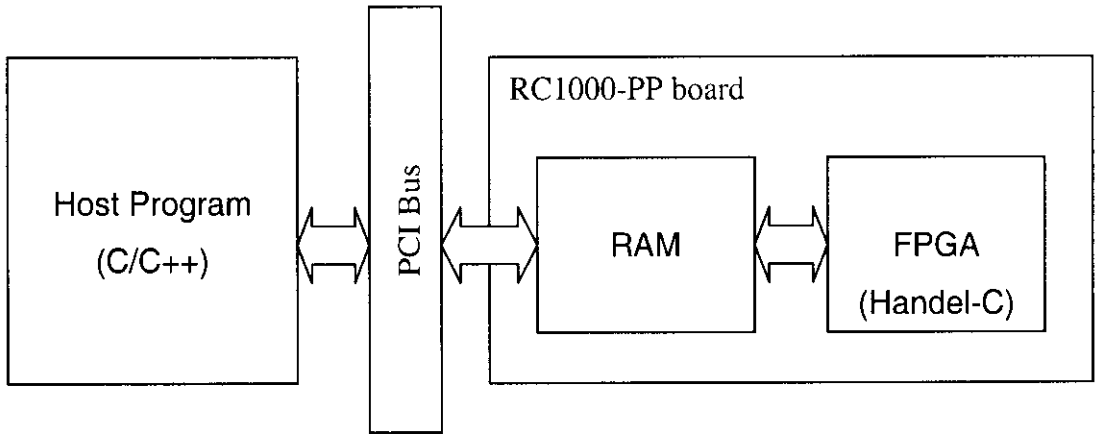

Figure 3.8 - Host-FPGA DMA of On-board Memory

An example of a typical system of software (host program in C/C++) and hardware (FPGA synthesized by Handel-C) that make use of the features discussed above for data transfer is shown in Figure 3.9.

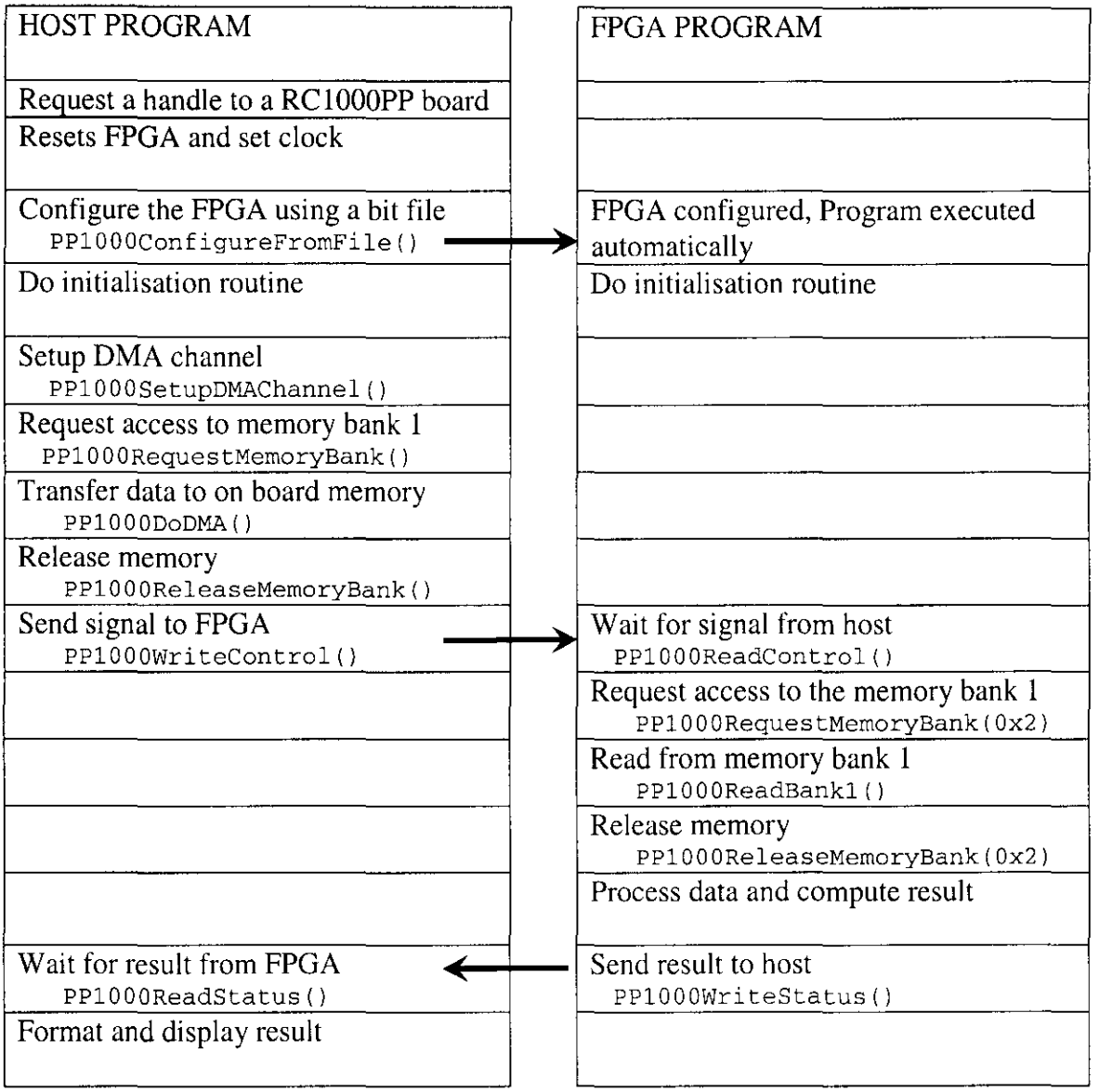

Figure 3.9 - An example of Host-FPGA Communication Process

3.9 Summary

In this chapter, the project requirements have been established. We have looked at the details of CAN data formats and touched on some important features of Handel-C. In next chapter, we will look at how the implementation of a simple CAN 2.0 A was designed.

4 Design of the CAN Network

This chapter describes the design of a CAN network that consists of three hardware nodes that will communicate with each other using CAN protocol. Messages will be transmitted in form of *CAN 2.0A standard dataframes*. Overall operation of the system will be controlled by software. Discussions will start with the approach, followed by the conceptual level design ideas and exploration of the viability of each design.

4.1 Approach

Basically, the execution of this project will not follow any one particular *Embedded System Design Methodology* — rather, ideas from a few methods will be considered. Some important factors that are considered when outlining the strategy of the execution are as follows:

- The aim of this project is to create a functional implementation with a reasonable degree of performance
- The objective is to study the implementation rather than produce a marketable product
- The architecture (FPGA) and tools (C, Handel-C) are pre-selected
- The use of reconfigurable device (FPGA) will makes it possible to optimise the whole system through refinement

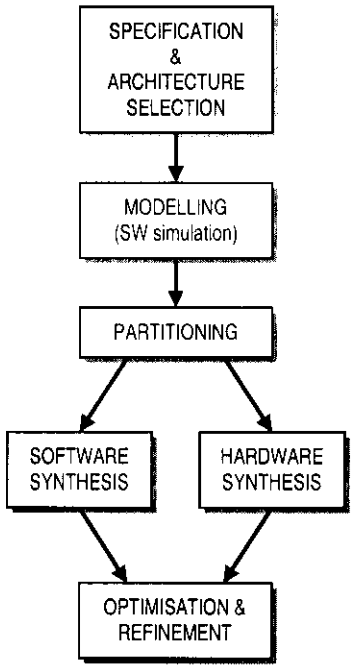

Figure 4.1 - Development Plan (Based on HW/SW Co-design Model)

- The use of Handel-C to programme the hardware means that it is possible and relatively easy to initially write the entire system in software, do partitioning and re-write the hardware parts.

The basic method will be based on conventional embedded system design methodology plus a few ideas taken from SystemC-based design flow (Figure 4.1). The outline of the methodology to be employed is as follows:

- **Specification** — based on project specification
- **Architecture Selection** — pre-selected, i.e. FPGA
- **Modelling** — a model of the network will be conceptualise at a high level using functional blocks, then refined to lower levels, converted to algorithms and flowcharts and finally written in C. The model will be entirely simulated in software on a PC. At this phase, the system will start as an *Untimed Functional* implementation and progressively refined to *Timed Functional*.

- **Partitioning** — those functions that perform low-level operations and those that have good potential for parallelism are prime candidates for hardware implementation.
- **Synthesis** — hardware implementation will be written in Handel-C and refined for parallelism. Software codes in C will be re-written to accommodate communication with the hardware.
- **Optimisation & refinement** — the whole system will be optimised for performance. Low level functions such as bus emulation will be refined.

4.2 High-level Design Ideas

Conceptually, the diagram in Figure 4.2 represents the network implementation. It consists of three CAN controllers (nodes) connected to a CAN bus. The CAN bus,

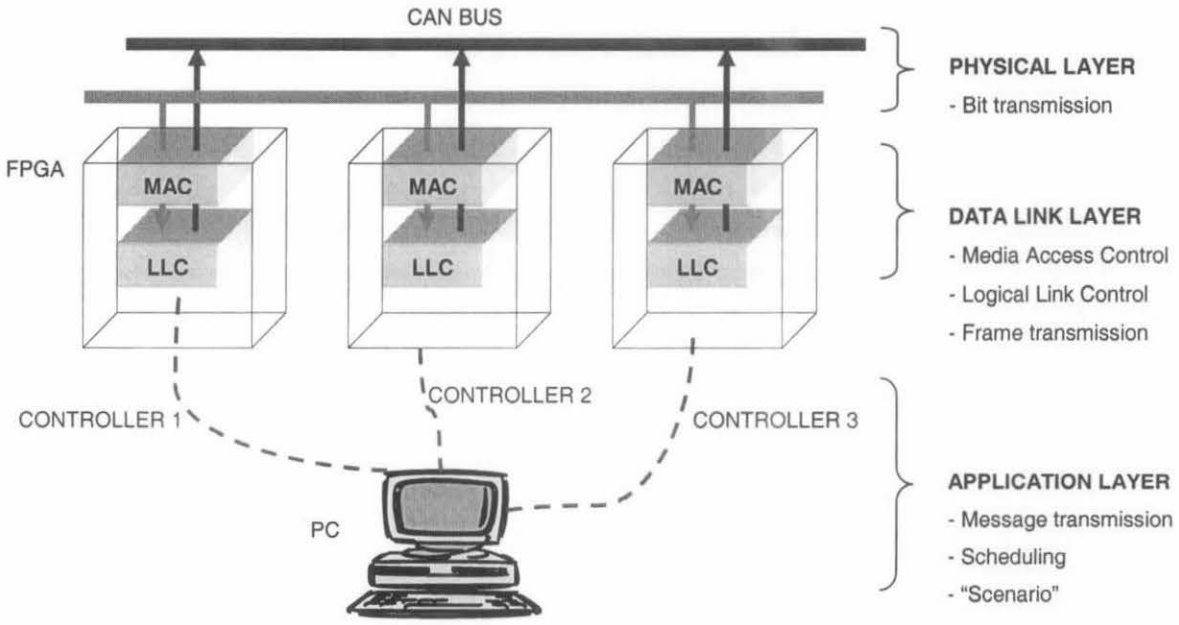

Figure 4.2 – The Proposed Network Logical Configuration

which forms the physical layer of the network, is essentially two pieces of wire. In the actual implementation, the bus will be simulated through hardware logics. However, the possibility of directly implementing the physical bus using hardware will be explored.

The data-link layer of the network will be implemented in hardware while the application layer will be in software so that there is a degree of conformance to the OSI layers in CAN model. This means that low-level functions such as frame transmission and frame-level error handling will be done here. All messages received will be raised to the application layer.

The application layer handles data at message level. It is essentially the heart of the network operation. In the actual implementation, the software will be written to handle the application layers for all three CAN controllers. It is likely to be run in an *executive cycle* i.e. a big loop that executes a series of small operations. Each of the controllers will take turn to send messages. A message sent by a controller is either acted upon or ignored by the other two — depending on the identifier of the message.

Although early partitioning has been done and the way the system is split was more or less determined, there was still a degree of flexibility in the configuration. Some functions such as dataframe formatting, CRC computation and error handling can be placed either in hardware or software. These options were explored during the development process.

4.3 CAN Bus Emulation

As stated earlier, the CAN bus will be emulated — although physical implementation will also be explored. Two viable options are message/token passing and function emulation.

4.3.1 Message Passing

In message passing (Figure 4.3), a message is passed from one controller to the next until it returns to its origin. The message originator sends the message to its

downstream neighbour, who in turn, passes the message to its downstream neighbour, and so on until the message arrives back to the message originator. The data frame could be modified slightly to accommodate a token field. The token can be used to store the originating controller number so that it will know when the message has returned. A

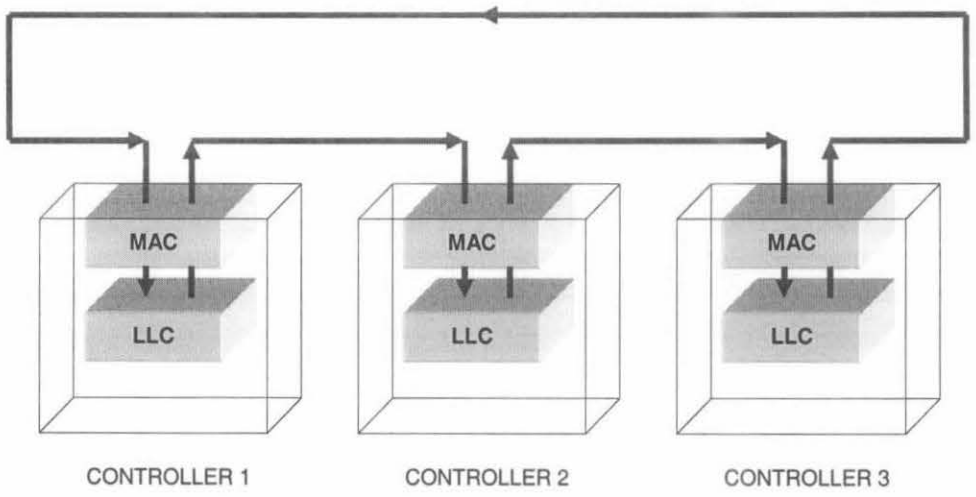

Figure 4.3 CAN bus emulation by message/token passing

message is destroyed when it returns. More than one controller can send messages at the same time. If the originator receives a different message than the one it sent out, it simply becomes a receiver.

Obviously, this technique does not mimic a CAN system correctly at the physical level. However it is good enough for the study of its functional behaviour. Alternatively, the message passing could be done at bit level — an option that will remain open for exploration.

4.3.2 Function Emulation

When the message passing bus emulation is verified to be working, the "bus" can be refined further by emulating its function only. In this option, a separate logical circuit device will be created to emulate the function of a CAN bus (Figure 4.4). Each

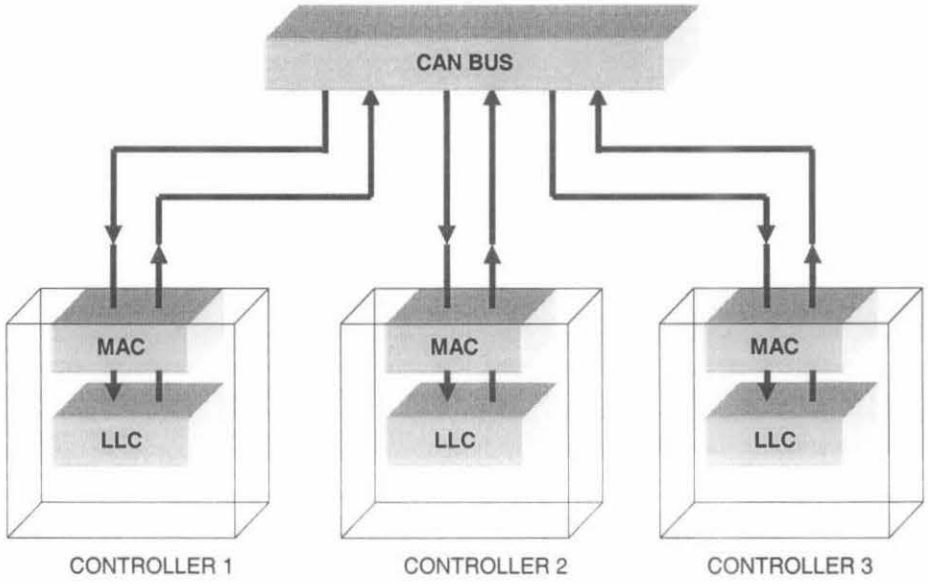

Figure 4.4 - CAN Bus Function Emulation

CAN controller will write bit data to the device and read from it as if it is a bus. The "bus" will also perform bit-level arbitration.

The arbitration process was planned to be implemented as described in Section 3.2. Being able to emulate a CAN bus at its function level will give a better understanding of the way the arbitration process is executed.

4.4 Arbitration Handling

When two or more CAN controllers start to write to the bus at the same time, an arbitration process will occur. When the controllers write to the bus, only a dominant value i.e. logic 0, get written to bus. A controller that has written a recessive value i.e. logic 1 to the bus will back off and becomes a receiver.

In actual CAN, each controller has to synchronised itself with the network's clock. Thus when they write a bit value to the bus, they do it at the same time. Every controller on the network operates a specified clock rate. However, the CAN designed

in this project has no notion of clock as in actual CAN. The controllers were not designed to work in parallel with other. Each of the controllers will be given a time slice to execute its tasks. The controllers will take turn to write a bit value to the bus or to read from it.

In order to emulate the arbitration process, the read and write process is separated into two phases that operate in cycles. In the first phase, each controller takes turn to write one bit value to the bus until every controller has done so. In the second phase, each controller takes turn to read one bit value from the bus. Then the two phases are repeated again. This is shown in Figure 4.5.

When a controller *writes* a bit value to the bus, that value is "logical-ANDed" with the bus value. For example, let say *Controller 1* wants to write a '1' to the bus; and the bus has value of '0'; thus the effective value written to the bus by *Controller 1* is a '0' because '1' AND '0' is '0'. Similarly, writing a '0' to a '1' will also result in a '0'. To make this sub-process work, the bus must be reset to logical 1 at the start of each *write phase*. At the end of the phase, if every controller has written a '1' to the bus, the bus value will remain as '1'. If any of the controllers has written a '0 to the bus, the bus value will change to a '0'.

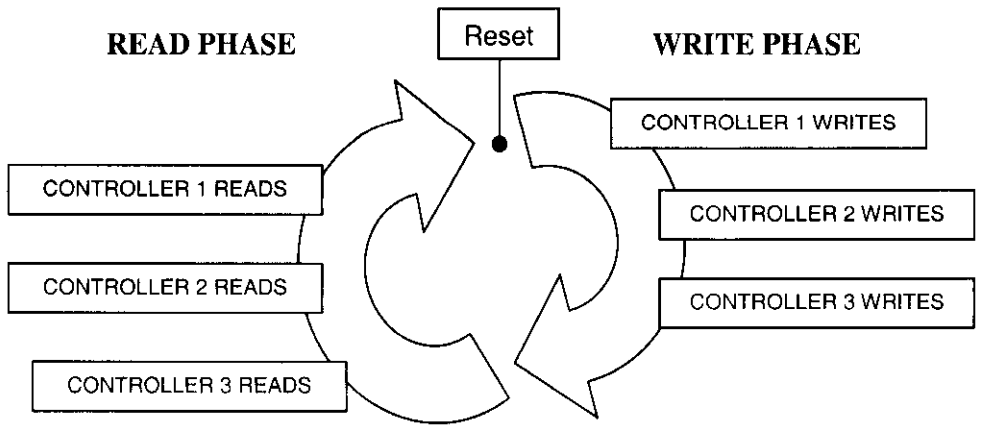

Figure 4.5 – The Write & Read Cycle

After a *write phase*, the controllers will enter a *read phase*, where every controller will take turn to read from the bus. Any controller that found that the value read from the bus is different from the value that it has written previously will know that it has lost the arbitration process. It will then has to back-off and becomes a receiver. If none of the controllers lost the arbitration, the "write and read" cycle will be repeated again for the next bit, until only one controller remain as the sole message sender.

Because of sequential nature of this scheme, it works well whether the program is software simulation or actually executed on the hardware. However by utilising the simple time-slicing technique the controller nodes appear to work in parallel. The process described above can be further enhanced with parallelisation. This technique will involve running Handel-C functions in parallel. Detailed discussion regarding this enhancement is presented in Chapter 7.

4.5 Hardware/Software Partitioning

In a any embedded system, some of the system's functions will be implemented in hardware while the rest will be in software. Some functions work better if implemented in hardware while some will work better in software. Conventionally, finding the optimum partitioning will take into account development time, the overall performance of the system, ease of use, code size and customisability. In this project however, the system will be partitioned so that at will closely resemble the OSI layers in CAN as described in Chapter 2 (Figure 2.4)

4.5.1 Partitioning strategy

The first implementation of the system will be done totally in software. Once a system that meets the functional requirements is produced, some of the functions will be moved to hardware. Those functions that perform low-level communication processing

will be the obvious choice because they contribute the most to performance bottlenecks. In this particular project, these are the functions that are located at the *physical layer*.

High-level functions i.e. those located at the *application layer*, will be left in the software. Other functions will either be moved to hardware or left in software depending on whether they contribute significantly to the overall performance of the system. Refinement and optimisation will be done until a satisfactory result is obtained.

4.5.2 Prototyping

The normal Handel-C program development was followed. The first step is to write a basic version of the program in C to test out the correctness of the algorithms. Several small programs can be written, each testing a certain algorithm or certain portions of an algorithm. When it is satisfied that the algorithms are correct, the C program is to be ported to Handel-C. Certain part of initial C program will be changed to conform to Handel-C or to take advantage of certain features not available in conventional C.

The Handel-C program can be sun in a simulation on the development PC without the need for an FPGA hardware board. When satisfactory result is obtained from the simulation, the necessary host program is written. The host program acts as a front-end to the FPGA program.

4.5.3 Software Simulation

The initial implementation of the main algorithms was written in C to verify their correctness. The main advantage of this approach is that C is more flexible compared to Handel-C in term of its type handling. In Handel-C, data of different width cannot simply be mixed in the same statement. Thus, writing in C allows the idea to be explored without worrying about the underlying data width.

The efficiency of the C compiler also means that several versions of the codes, each with different approach to the same idea, can be tested rapidly. And lastly, because C/C++ has an extensive list of input output functions compared to Handel-C, debugging by data comparison can be done easily.

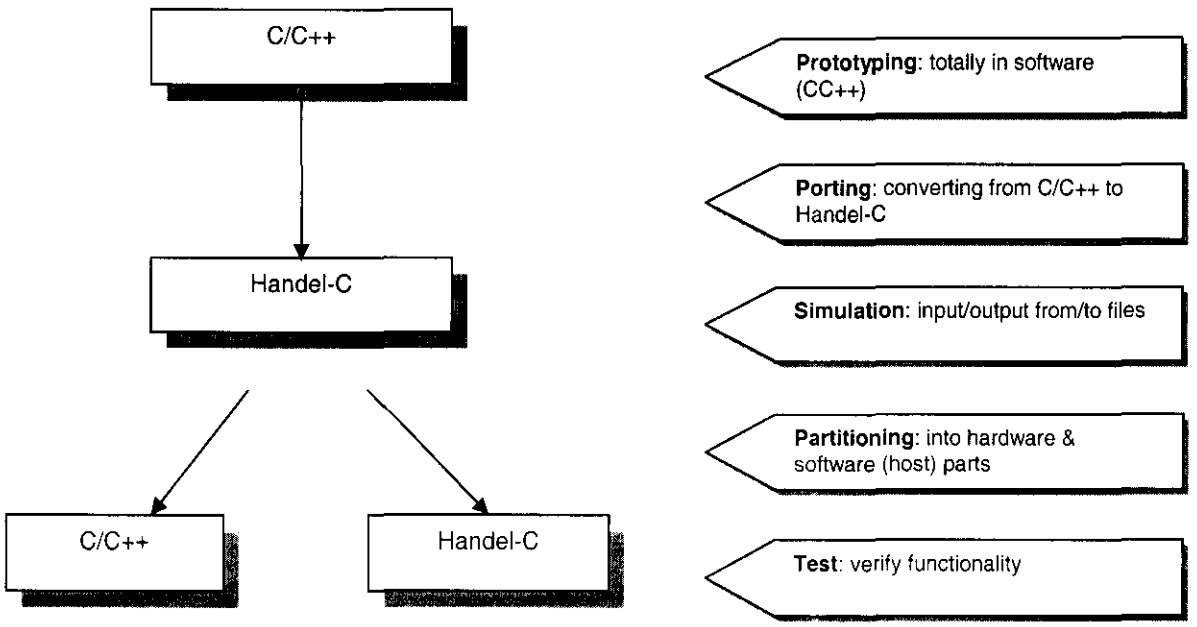

Figure 4.6 Hardware/Software Partitioning Strategy

4.5.4 Porting C to Handel-C

Once the basic idea was successfully implemented in C, the program was ported to C. When porting from C to Handel-C the following main tasks were done:

- specifying data widths
- breaking certain compound expression into simple statements
- modifying codes to take advantage of Handel-C features

In the first task, most of the data of type *int* were converted to *unsigned int 8*. Several variables whose values can exceed 255 were converted from *int* to *unsigned int 16*. Several data were converted to their exact width as in the CAN specification. As an

example, four important variables that correspond to four fields in the CAN dataframe are defined as follows:

```
#define  nCTRL 3

unsigned int 12 arbit[nCTRL];          // Arbitration field
unsigned int 6  control[nCTRL];        // Control field
unsigned int 8  data[nCTRL][8];        // Data field
unsigned int 16 crc[nCTRL];            // CRC field
```

Being able to set the data widths to exactly the same widths as used in an actual CAN dataframe has one main advantage. The data can be manipulated without worrying about extra unused bits. In conventional C, the width of a data cannot be arbitrarily set non-standard values of 8 (for char), 16 (for int), 32 (for float) or sometimes 64 bits. There are extensions in Microsoft C++ to set *int* to 8 bits but this is not standard.

4.5.5 Hardware Simulation

The DK1 development environment includes several features to make system development easier. One of the features is the ability to run a simulation of a Handel-C program on a PC without the need for an FPGA board. To run a program in a simulation, its *build configuration* is set to *debug* [28].

The main limitation of the simulation mode is the lack of proper input and output constructs. Input and output are handled through channels. An input channel can be defined using the *chanin* keyword and output by using *chanout*. An input file must be specified for every *chanin* definition while for *chanout*, a file is optional. During simulation, the program will read from the input files specified and send its output to the output files. If no output files were specified, the output will be displayed in the debug window of the DK1 development environment.

Each file can only be assigned to one input/output channel and each channel can only read one data per line. Thus, an input file normally consists of a column of numbers each on a line of its own. The same limitation applies to the output files.

To overcome this limitation, sample data input was hard-coded into the program and the outputs were written into several output files. These files were then inserted into a pre-formulated spreadsheet so that more effective debugging could be made. Using the spreadsheet, values of various counters, status and other data could be verified against each other. When incorrect values were found on the spreadsheet, corrections were made in the program codes and the process was repeated. These were done until a satisfactory result was obtained. A copy of the output spreadsheet is in the appendix.

4.6 Summary

The basic idea discussed was to create a network of three CAN controller nodes in an FPGA. Basic communication protocol was to be built inside each node while the whole operation of the system is to be controlled by software. Two inter-node communication techniques were discussed (message passing and bus function emulation) and function emulation was found to be a more accurate representation of an actual CAN bus. The implementation of this design is described in next chapter.

5 Implementation

This chapter presents the end product and describes the various components of the system and how they interact with each other. A walkthrough of a typical operation of the system is presented. Critical functions of the programs and how they work are also discussed.

5.1 The System

The system consists of a hardware subsystem and a software subsystem. Both are essentially made up of one program each. The program for the software subsystem is referred to as the *host program* and the hardware program is referred as the *FPGA program*. The host program runs on a PC and is written in C++. It was developed as a command prompt mode program. The FPGA program is written in Handel-C.

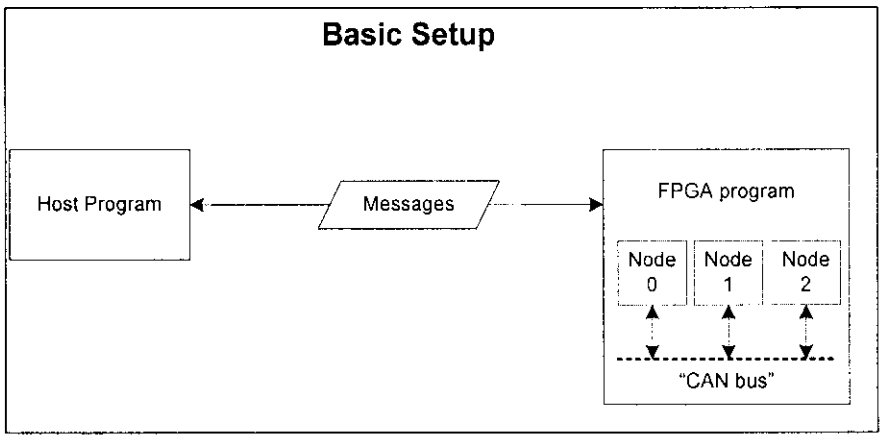

Figure 5.1 - Basic Setup of the Host and FPGA programs

The host program plays the role of the *application layer* in the OSI Reference Model while the hardware acts as the *Data Link* and *Physical* layers of CAN. The application layer in the host program communicates with three virtual CAN controller nodes (data link layer) in the FPGA as shown in Figure 5.1. The nodes are connected to a virtual CAN bus (physical layer). The CAN controller nodes are executed as virtually independent units by the FPGA programs. The FPGA program acts as a sort of intermediary between the host and the nodes.

5.1.1 Walkthrough

When the system is started, it initialises the variables and hardware and presents the user with a simple menu as shown in Figure 5.2. As an example, an operation of sending a message from the first node will be discussed. This operation is done by pressing selecting option 1 from the menu.

When key 1 is pressed, the host program will send a message to the FPGA. The host message contains the number of the recipient node (in this case, *node 0*), a message identifier, length of the data and the data itself. This information will be displayed on the screen (Figure 5.3). The receiving node will process the host message to convert it into a suitable format i.e. a *dataframe* (Figure 3.1) to be written to the CAN bus. It this point, the node will change its mode into a *sender*.

Other nodes in the network, upon detecting a message sent by *node 0*, will change their mode into receiver and start reading the message into their own *dataframe*. When the message transmission is completed, each receiver will send a message to the host program. The messages received will be displayed on the screen (Figure 5.4). It can be verified in a successful transmission that the host messages sent and received would be identical.

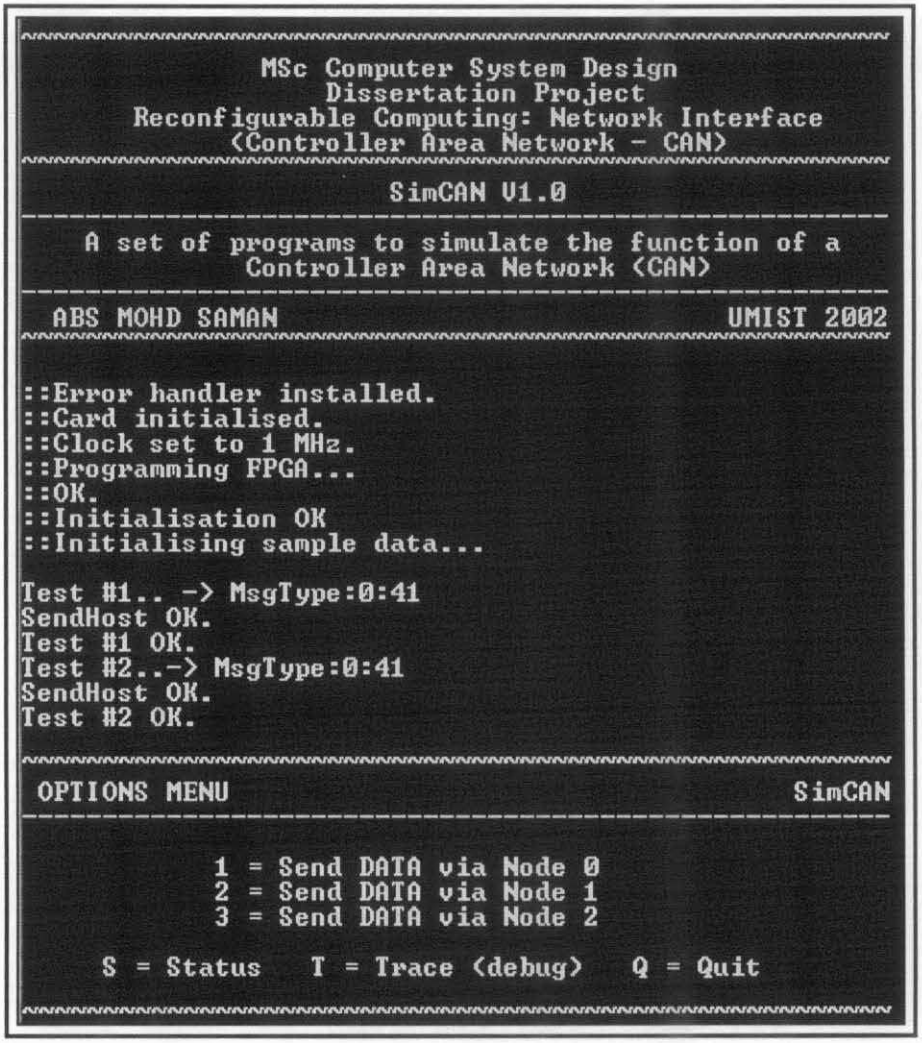

Figure 5.2 -Host Program's Menu

Figure 5.3 - Host sends a message to Node 0

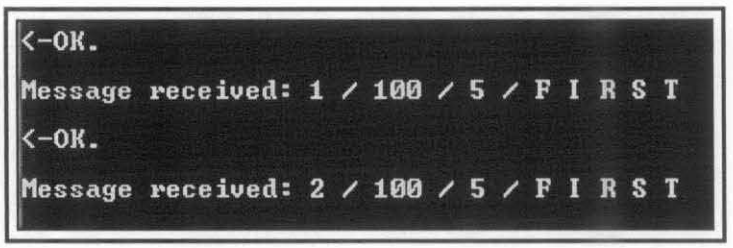

Figure 5.4 - Host receives messages from Node 1 & 2

At any time, the user can press any of the option keys in order to execute intended operation. Table 5.1 lists all the available option keys and their operations. Figure 5.5 shows an example how status reports were received when the **S**-key was pressed several times. The trace mode can be toggled on and off by pressing the **T**-key. This was used extensively during debugging. The program can be ended by pressing the **Q**-key. When it ends, the program sends an appropriate message to the FPGA and closes the RC1000-PP board.

Key Pressed	Action
1	Sends a message to Node 0. Node 0 will write the data to the CAN bus
2	Sends a message to Node 1. Node 1 will write the data to the CAN bus
3	Sends a message to Node 2. Node 2 will write the data to the CAN bus
S	Requests status report from FPGA
T	Toggles trace(debug) mode
Q	Ends the programs

Table 5.1 - Host Program Responses to Key Presses

```
-> MsgType:5:5
SendHost OK.

Status received from CAN nodes...

        Node    Mode    Bit#    Write   Read
        #0      SEND    8       0       0       [88,0]
        #1      RECV    8       0       0       [0,88]
        #2      RECV    8       0       0       [88,88]


-> MsgType:5:5
SendHost OK.

Status received from CAN nodes...

        Node    Mode    Bit#    Write   Read
        #0      SEND    9       0       0       [88,0]
        #1      RECV    9       0       0       [0,88]
        #2      RECV    9       0       0       [88,88]


-> MsgType:5:5
SendHost OK.

Status received from CAN nodes...

        Node    Mode    Bit#    Write   Read
        #0      SEND    10      1       1       [88,0]
        #1      RECV    10      0       1       [0,88]
        #2      RECV    10      0       1       [88,88]
```

Figure 5.5 - Getting Status Reports from FPGA

5.2 The Host Program

During execution, the program on the PC acts as a host to the FPGA program. Every input into and output from the FPGA program goes through the host. When started, the host program loads the FPGA program from its *bit file* into the hardware using the following function call:

```
PP1000ConfigureFromFile(Handle, "canfpga.bit");
```

In the above statement, `PP1000ConfigureFromFile()` is one of library functions of the RC1000-PP board. `"canfpga.bit"` is the name of the bit file that was built from the Handel-C program written for the FPGA. `Handle` is the *handle* obtained when opening an RC1000-PP card by using such command as: `PP1000OpenFirstCard(&Handle);`

The FPGA program will then start automatically. As a control measure, the FPGA program was written so that at the beginning the execution it will wait for a signal from the host before executing its main routine.

The host program controls the overall operation of the CAN controller nodes. When it wants to send some data through node 0, it will send a message to the FPGA program. The message contains a node number, a message ID and the data itself. When the FPGA program receives a message from the host, it will store it and flag the intended node. The host then will wait for the FPGA program to return another message indicating that the data has been successfully sent and received.

To a user, the host program offers a menu of commands as listed in Table 5.1. It continually waits for the user to press a command key. When a command key is pressed, it will act accordingly. On most commands, the host program will send a message to the FPGA (Figure 5.6). The type of messages that can be sent to the FPGA is listed in Table 5.2.

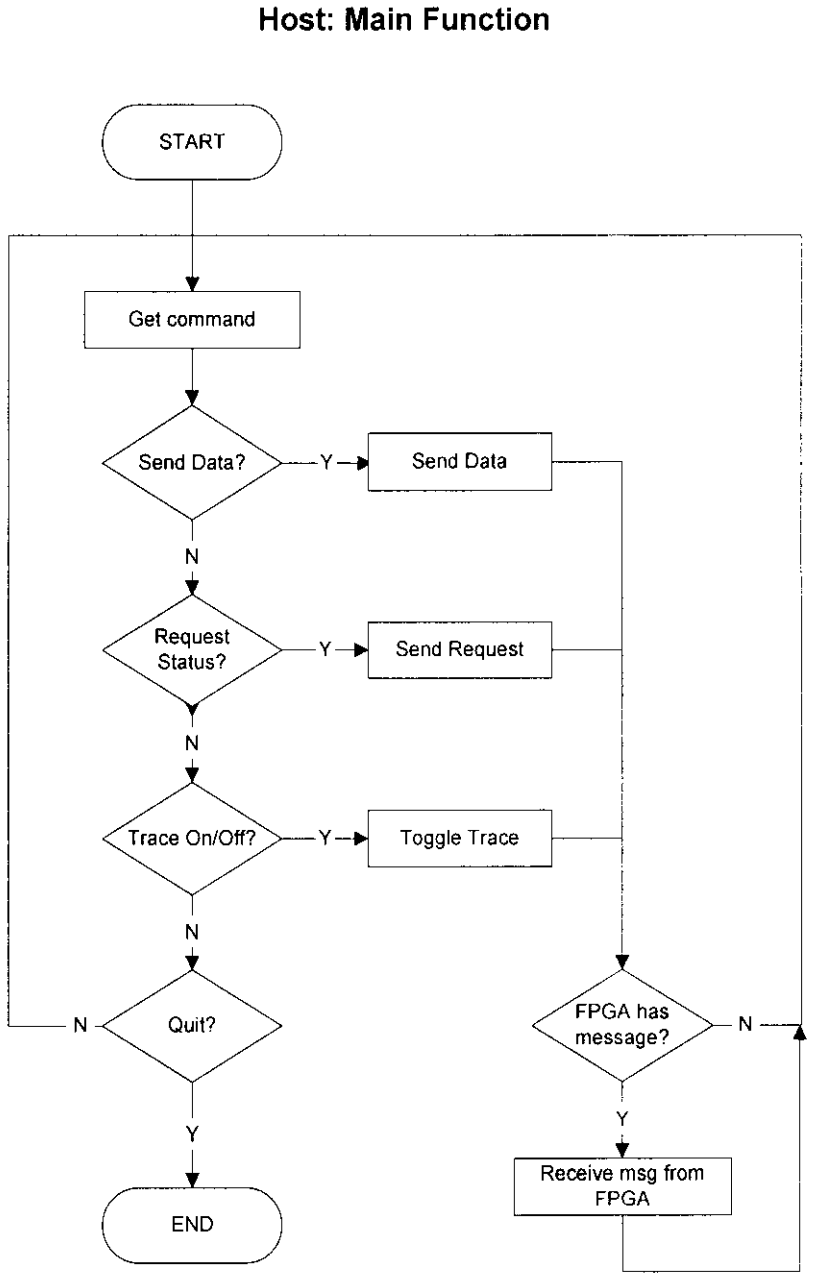

Figure 5.6 – Host's Main Flowchart

At the same time, it also waits for any messages from the FPGA program. There are only two types of messages sent by the FPGA: DATA and TEST. When a message of type DATA is received, its contents will be displayed in the following format:

```
c / id / n / data 0, data 1, … , data n-1
```

In this format, **c** is the controller node number, **id** is an identifier for the data, **n** is length of the data in number of bytes; and **data 0** to **data n-1** are the actual data. Each component of the message is one byte in size. As discussed earlier in section 3.1.2, the identifier in a dataframe is 11 bits in size. Thus, during conversion from a host message to a dataframe (and vice versa), the first three MSBs of the identifier are ignored.

TYPE	DESCRIPTION
DATA	Contains a message to be sent into the CAN bus
STATUSREQ	Request status report from the FPGA
TRACEON	Switch on "trace" mode
TRACEOFF	Switch off "trace" mode
TEST	Used to test host-FPGA communication and for debugging
ENDPRG	Signals end of program: close the hardware and end program

Table 5.2 - Types of Messages from Host to FPGA

5.2.1 Functions

The followings is a list of all the functions in the host program and their descriptions:

- `Handler()` – handles errors related to access to the RC1000PP board.
- `initialise()` — initialises the RC1000PP board i.e. installs the error handler, open the first RC1000PP card available, sets clock rate and loads bit file.
- `fpga_has_message()` — checks whether the FPGA program has a message to send.
- `setup_message()` — prepares the message to be sent to the FPGA in a suitable format.
- `display_message()` — displays the contents of message received from the FPGA.
- `receive_fpga()` — receives message from the FPGA.

- `send_fpga()` — sends message to the FPGA.
- `main()` — main function of the program; contains one main loop that checks for key press from the user and acts accordingly. It responses to certain inputs as listed in Table 5.1

HOST	FPGA
Main function	
Start Load FPGA program (.bit) Wait for FPGA to get ready	Start Tell Host: "Ready"
Loop Read user input If "send message" Send message to FPGA If "request status" Receive status from FPGA If FPGA has message Receive message from FPGA Display message Repeat	Loop Do write Controllers 0, 1, 2,... Do read Controllers 0, 1, 2,... Write status Repeat
Host sends message to FPGA (node)	
If have message to send Tell FPGA: "Have Message" Send message to FPGA	Query host for message If host has message Get message from host Node creates data frame Node checks bus whether it is free If bus is free Node writes dataframe to CAN bus
Host receives message from FPGA	
Query FPGA for message	Node scans CAN bus If other node is writing Node reads from bus into dataframe Node writes ACK to bus Tell host: "Have message"
If "has message" Receive data from FPGA Tell FPGA: "OK" Display information	Wait for "OK"

Table 5.3 - Host - FPGA interaction

When the programs are in trace mode, the execution of FPGA program is moderated by the host program. At each read/write cycle, the FPGA will wait for a message from the host. If the user chooses to request a status at every read/write cycle, detailed changes in the CAN nodes can be verified. This is useful for tracing and extensively used during debugging.

A summary of major tasks executed by the host program is listed in Table 5.3 in form of algorithms for their relevant functions. It also lists similar information for the FPGA program. Essentially, the table shows how the two programs interact with each other.

5.3 The FPGA Program

The FPGA program is essentially made up of two parts: *host communications* and *bus communications*. The host communications part consists of several functions that handle the necessary communication with the host program. This involved a lot of function calls to the RC1000-PP library. The bus communication part consists of several other functions that handle writing to and reading from the virtual CAN bus. This involved a lot of bit manipulations and computation of data of various sizes. The functions for bus communications are discussed in the following sub-section while those for host communications will be described in Section 5.4.

5.3.1 Writing to the Bus

Two of the most critical tasks of a CAN controller node are writing to the bus and reading from it. In this implementation, these tasks are handle by two functions called `controller_write()` and `controller_read()`. These two functions are executed repeatedly in a cycle of two phases: *write phase* and *read phase*; very similar to what has been described in Section 4.4 (see also Figure 4.5). In each phase only one

bit of data is written to or read by any particular node. Executed repeatedly at a very high speed, this gives an appearance that the nodes were running in parallel. This technique is similar to *time-slicing* but without a fixed time allocated for each process. A flowchart (some minor tasks have been omitted for clarity) for the main function of the FPGA is shown in Figure 5.7.

The `controller_write()` function checks the mode of the CAN node and take certain actions as summarized in Table 5.4. The main task of the `controller_write()` function is to write each bit in a frame to the bus. It will also handle other tasks depending on the mode it is in. If it is in a *receive* mode, it will check if it is pointing at the ACK slot. If so, it will verify the data using CRC calculation. If no error is detected from the calculation, the function will write an ACK to acknowledge that correct data has been received. If the CAN node is in *idle* or *wait* mode, it will do nothing.

MODE	Controller_write()
IDLE	Do nothing
RECEIVE	If ACK slot then write ACK
SEND	Write frame to bus
WAIT	Do nothing

Table 5.4 - Tasks of controller_write() function

If a node is in *send* mode, it will call the write_frame() function which will write the contents of its dataframe to the bus one bit at a time. Each time the function is called, it will write one bit and increase its internal counter. While writing, it also does bit-stuffing by calculating the number of consecutive identical bits. If the number of identical consecutive bits is more than five, a stuff bit will be inserted.

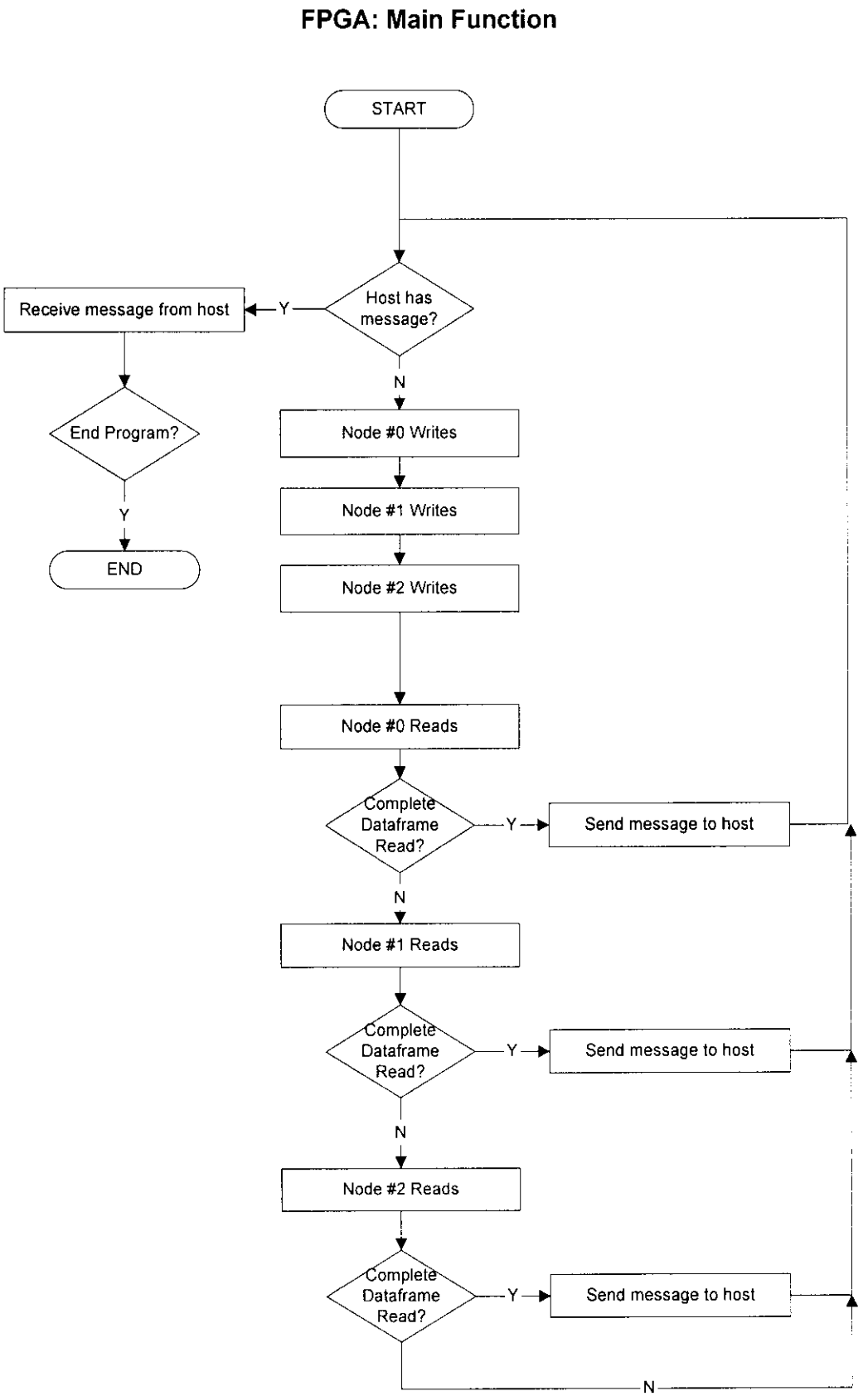

Figure 5.7 – FPGA's Main Flowchart

A 'sending' controller node will also read back the value that it has written to the bus. If it is not the same as the value that has written to the bus, that it knows that either it has lost an arbitration (if the bit was from an arbitration field) or there was an error while writing (if the bit was outside the arbitration field). If it has lost in an arbitration process, the node will become change its mode to wait. In this mode, it becomes a *receiver* while waiting the bus to be free and available for a re-transmission (Table 5.5).

A 'waiting' controller node will know that the bus is available when it detected idle time of more than the idle threshold of 10 bits. When it detected that there were 10 consecutive recessive bits (logic 1) on the bus, it will change its mode to *send* and start transmitting its dataframe again. It will go through the arbitration process once again. These steps are repeated until its dataframe has been successfully transmitted.

5.3.2 Reading from the Bus

List of tasks performed by the `controller_read()` function is listed in Table 5.5. Its main task is to read the bit of the *dataframe* from the bus. However, if it is in *idle* mode, it will check whether other nodes are transmitting any data onto the bus. It does so by scanning the bus for a Start of Header (SOF). An SOF is identified by a pattern of 10 ones followed by a zero (`11111111110`).

A node that was in idle mode, upon detecting an SOF will change its mode to *receiver*. In this mode, the node reads from the bus one bit at a time, removes any stuffed bits (de-stuffing) and add the remaining bits into a dataframe.

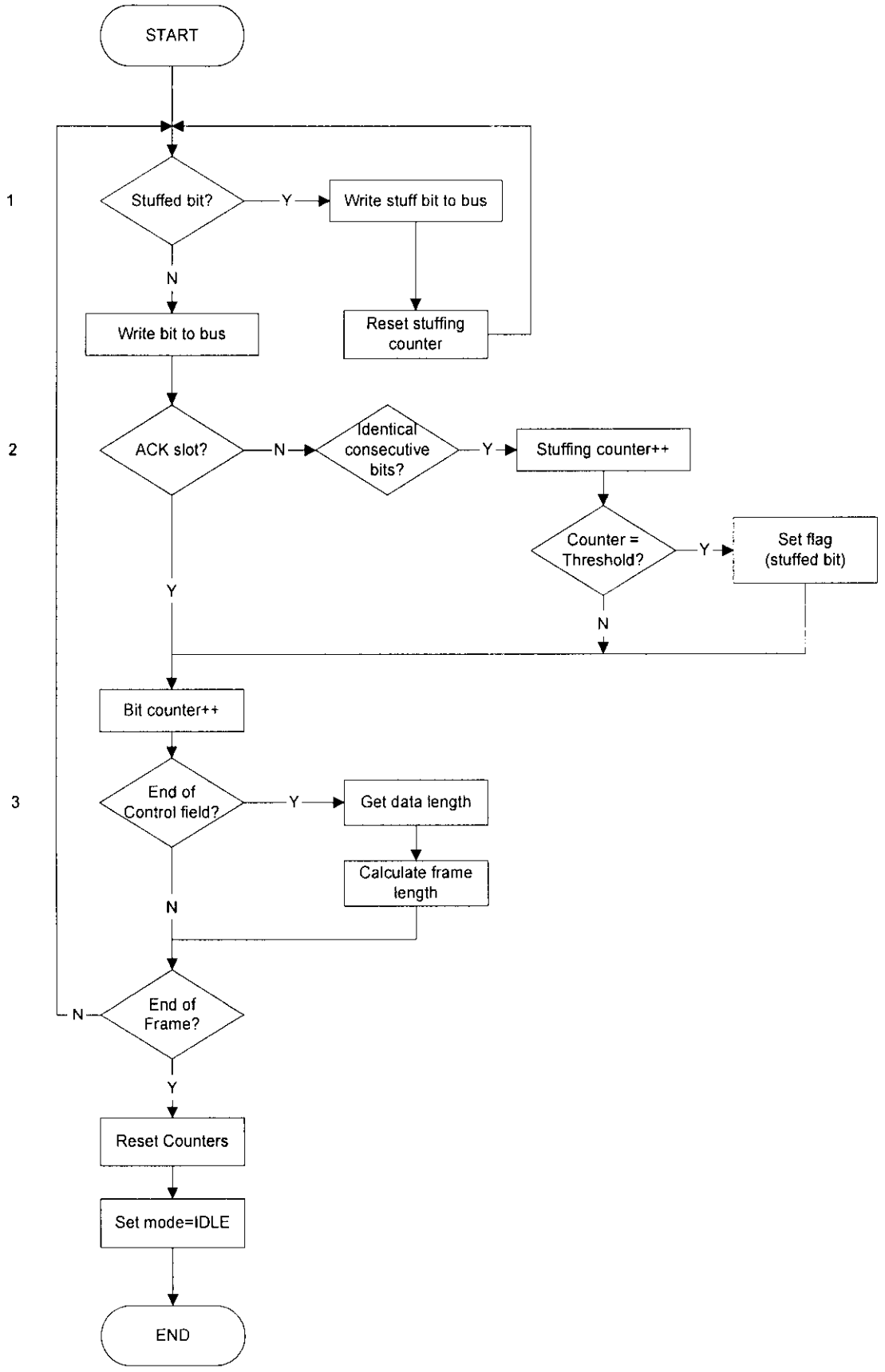

Figure 5.8 – Flowchart for write_frame() function

While reading from the bus, the node also checks if what it has written to is the same as what it is reading. It also calculates the length of the frame and writes into the ACK slot (Figure 3.1) near the end of the frame. Figure 5.9 shows a flowchart for the `read_frame()` function which handled the operation described above.

MODE	controller_read()
IDLE	Scan for SOF (Start of Header) If SOF found then Read Frame from bus Else query for data from Host
RECEIVE	Read Frame from bus
SEND	Read Frame from bus If writing ARBITRATION field, compare written/read bits If written bit <> read bit Change mode to WAIT
WAIT	Read Frame from bus Detect whether bus is free If bus is free Change mode to WRITE

Table 5.5 - Tasks of the controller_read() function

5.4 Host – FPGA communications

Communications between the host program running on the PC and the FPGA program running on the Celoxica RC1000-PP board were be done by using library functions supplied with the board. The host can send data to the board and vice versa in three modes: 1 bit, 1 byte and direct memory access (DMA).

The bit-size data transfer is handled by the following library functions:

Functions used in the host program:

- `PP1000SetGPO()` – set the GPO (general purpose output) pin
- `PP1000ReadGPI()` – read the status of the GPI (general purpose input) pin

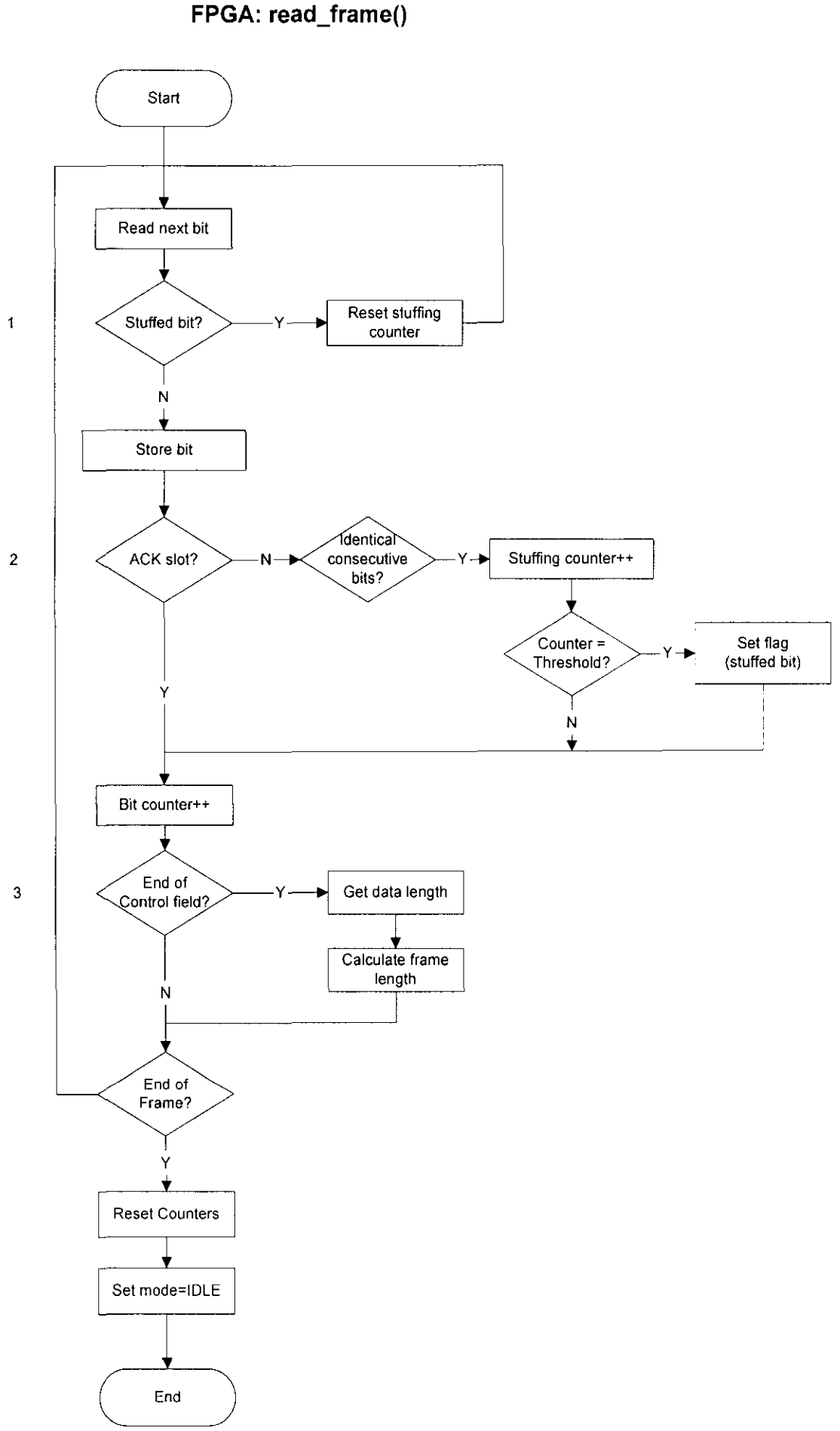

Figure 5.9 - Flowchart for read_frame() Function

Functions used in the FPGA Handel-C program:

- `PP1000ReadGPO()` – read the status of the GPO pin
- `PP1000SetGPI()` – set the GPI pin

These functions were used by the host program to send an alert to the FPGA program or vice versa. For example, the FPGA can be programmed to check the status of the GPO pin at regular intervals using the `PP1000ReadGPO()` function. When the host wants to get the attention of the FPGA it sets the GPO pin to high using the `PP1000SetGPO()` function. Upon detecting this, appropriate actions are taken by the FPGA program.

The byte-size data transfer is handled by another set of library functions:

Functions for the host program:

- `PP1000WriteControl()` – send one byte of data to the FPGA
- `PP1000ReadStatus()` – receive one byte of data from the FPGA

Functions for the FPGA Handel-C program:

- `PP1000ReadControl()` – receive one byte of data from host
- `PP1000WriteStatus()` – send one byte of data to host

When the host sends a data to the FPGA by calling a `PP1000WriteControl()` function, the FPGA receives it through a corresponding `PP1000ReadControl()` function call. Multiple bytes can be sent and received by having an appropriate number of function calls in the host program and the same number of corresponding function calls in the FPGA program. This method is used for sending messages from the host to the FPGA and vice versa.

One common property of this set of functions is that if for example the FPGA is expecting a byte of data from the host through the `PP1000ReadControl()` function, it will not continue until the data is received. This particular property of the functions is

useful in adding a *trace* operation to the system where the host needs to send "stop and start" signals to the FPGA. The other pair of functions (`PP1000ReadStatus()` and `PP1000WriteStatus()`) also behave in a similar way and were utilised in similar manners.

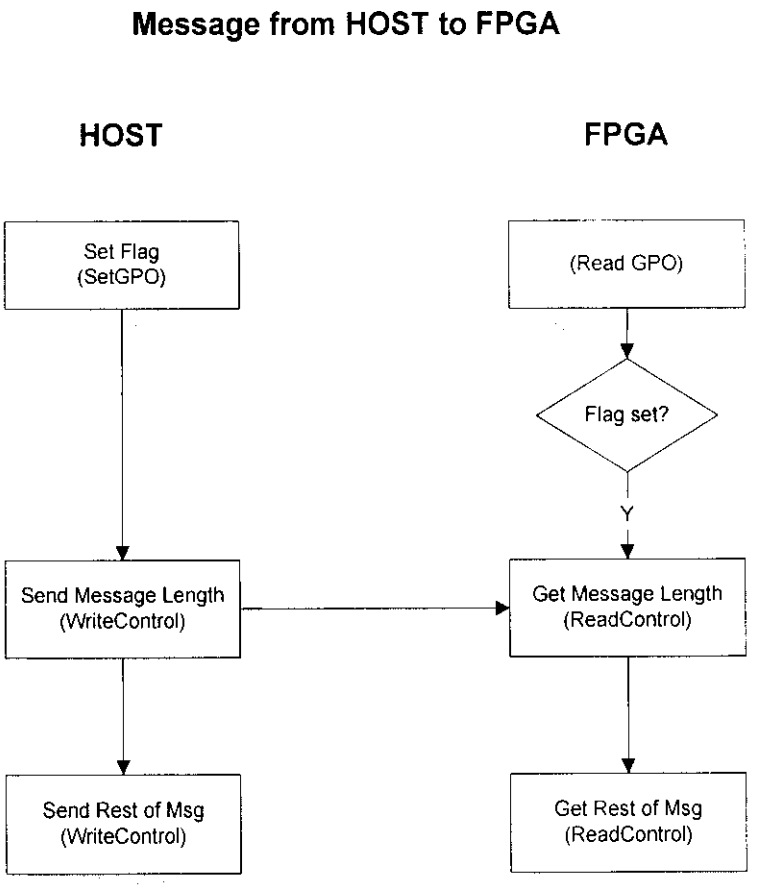

Figure 5.10 - Sending Message from Host to FPGA

The third mode for data transfer, the DMA, is handled through the following set of library functions which allow direct access to the onboard memory blocks of the RC1000-PP card [ ]:

Functions for the host program:

- `PP1000SetupDMAChannel()` – set up a DMA channel
- `PP1000RequestMemoryBank()` – request access to a memory bank
- `PP1000DDMA()` – execute the DMA data transfer

- `PP1000ReleaseMemoryBank()` – relinquish access to a memory bank
- `PP1000SetupDMAChannel()` – close the DMA channel used

Functions for the FPGA Handel-C program:

- `PP1000RequestMemoryBank()` – request for access to a memory bank
- `PP1000WriteBank()` – write to a memory bank
- `PP1000ReadBank()` – read from a memory bank
- `PP1000ReleaseMemoryBank()` – relinquish access to a memory bank

5.4.1 Host-FPGA Message Transmission

The set of commands for bit-sized and byte-sized data transmission described above were utilised for sending messages from host to FPGA. All message types described in Table 5.2 were sent this way. When the host wants to send a message to the FPGA, it will alert the FPGA by calling a `PP1000SetGPO()` function and pushing the first byte of the message by calling a `PP1000WriteControl()` function. The FPGA, upon detecting this signal, read the first byte by calling a `PP1000ReadControl()` function. The first byte contains the length of the rest of the message in number of bytes. The host will subsequently push the rest of the data to the FPGA byte by byte using the same `PP1000WriteControl()` function as many times as needed. The FPGA read the bytes by calling the `PP1000ReadControl()` function the same number of time. A flowchart for this process is shown in Figure 5.10.

The process of sending a message from the FPGA to the host was handled in the same manner but by using the `PP1000SetGPI()`, `PP1000ReadGPI()`, `PP1000WriteStatus()` and `PP1000ReadStatus()` functions.

The DMA functions were used for status reporting. The FPGA will write critical values such as bit counts, modes of controller nodes etc to a memory bank at every write/read cycle. Because of the way the function set work, the host can read these values at any time without interrupting the controller nodes.

5.4.2 Data Buffers for Host-FPGA communication

A message is transmitted from the host to FPGA and vice versa in a data buffer. The buffer is twelve words long and its format is shown in Figure 5.11. The same buffer is also used for transmitting test messages between the two sub-systems. Variables to accommodate the buffer were defined as an array of *char* in the host's C/C++ program; and as an array of *unsigned int 8* in the FPGA's Handel-C program. During debugging, the size of the buffer has been increased to accommodate the test data transmitted.

No:	0	1	2	3	4	5	6	7	8	9	10	11
	Node#	Msg ID	Data length	Data 0	Data 1	Data 2	Data 3	Data 4	Data 5	Data 6	Data 7	Data 8

Figure 5.11 - Message Buffer Format

When the host wants to send a message to the FPGA, it will set up the buffer by calling a function named `setup_message()` (see Listing L1 in the appendix). This function writes vital information into the buffer array so that it is ready for transmission. When the buffer is set up, it is sent to the FPGA through a function named `send_fpga()` (see Listing L2 in the appendix). This function sends the contents of the buffer one byte a time to the FPGA. Sending a message from FPGA to host is also handled in a similar manner.

At the end of each write/read cycle, the FPGA stores vital information into a status buffer. The format of the buffer is shown in Figure 5.12. This information is read by the host program if so requested by the user and displayed on the screen.

No:	0	1	2	3	4	5	6	7	8	9	10	11
	Mode 0	Bit Count 0	Written Bit 0	Read Bit 0	Mode 1	Bit Count 1	Written Bit 1	Read Bit 1	Mode 2	Bit Count 2	Written Bit 2	Read Bit 2
	Node 0				Node 1				Node 2			

Figure 5.12 - Type 2 buffer contents (CAN nodes status)

5.5 CRC Calculation

To verify that any frame sent out by a transmitter is received correctly by a receiver, cyclic redundancy checking (CRC) is employed. In order to carry out the CRC calculation, the divisor is defined as the polynomial. The coefficient of the polynomial is given by the de-stuffed bit stream consisting of Start of Frame, Arbitration Field, Control Field and Data Field (if present). The bit stream is padded with 15 zeroes. This bitstream is divided with the following polynomial:

$$X^{15} + X^{14} + X^{10} + X^{8} + X^{7} + X^{4} + X^{3} + 1$$

Bit 15 of the polynomial can be ignored in the actual calculation. Thus the polynomial value used is 4599hex. The remainder of this polynomial division is the CRC Sequence transmitted over the bus. In order to implement this function, a 15-bit shift register is used. The following algorithms (Figure 5.13) [8] was referred to when coding the CRC calculation for the CAN controllers. Written in Handel-C, the equivalent codes are as shown in Listing L3 in the appendix.

Portion of the dataframe that was included for CRC calculation are the *arbitration, control* and *data* fields. The result of the calculation is appended as a CRC field located after the data field.

```
CRC_RG = 0;
REPEAT
        CRCNEXT = NXTBIT EXOR CRC_RG(14);
        CRC_RG(14:1) = CRC_RG(13:0);
        CRC_RG(0) = 0;
        IF CRCNXT THEN
                CRC_RG(14:0) = CRC_RG(14:0) EXOR (4599hex);
        ENDIF
UNTIL( CRC SEQUENCE starts or there is an ERROR condition )
```

Figure 5.13 - CRC Calculation Algorithm

Several other established methods of CRC calculation using lookup table have also been considered but found to be unsuitable because they are based on 8, 16, 32 or 64 bit polynomial while CAN uses a 15 bit polynomial in its CRC calculation.

5.6 Summary

The implementation consisted of two programs: The Host (written in C/C++) and The FPGA (written in Handel-C). Communications between the two programs were handled by several library functions supplied with the RC1000-PP card. The two OSI layers in CAN were totally implemented in the FPGA (hardware), closely resembling an actual implementation of CAN. Testing and verification of the system is described in next chapter.

6 Evaluation/Testing

This chapter looks at the final artefact in light of the requirements identified in Chapter 3. Three stages of evaluation are considered: internal, external and meta-level. The internal stage will examine to what extent does the system produce the expected results. The external stage will look at how much does it satisfy the requirements discussed in Chapter 3. The meta-level stage will reflect on the process/method involved in performing the investigation.

6.1 Internal Evaluation

The behaviour of the system is like a set of real CAN controllers communicating with each other. Messages sent through one controller node are successfully received by another. The controller operates at a very high speed, a message will only take a fraction of a second to travel from one node to another.

Virtually, there are three CAN controller nodes in the hardware. Although physically, this is not how real CAN is set up, nevertheless, the nodes mimic the exact behaviour of real CAN controllers. Ideally, each node should be in its own hardware and connected to each other by a pair of wires.

Overall, the system performs as expected and produces expected results. When a message is sent from one controller, the other controllers will receive it correctly. When more than one controller try to send messages at the same time, they will go into an

arbitration process. The arbitration process has been verified to work correctly in the simulation mode and can be traced in the actual program.

Much of the data produced in the FPGA program is used internally by the program itself. It has been found that the most effective was of verifying that the values produced are correct are by tracing the values in the simulation mode. In this mode, data can be written to files. Due to the fact that the simulator allows only one series of values to be written to one files, several files were created. These files were then merged into a spreadsheet.

In the spreadsheet, critical values are listed side by side. They compared to each other and how each of the series changes values are also verified whether it is correct or not. A copy of the spreadsheet is available in the appendix.

6.2 External Evaluation

The system was built to consist of two CAN layers and one OSI layer. The CAN layers are: *Data Link* layer and *Physical* layer. The OSI layer is *Application* layer. In this implementation the function of the *physical* layer was simulated but the functions of the other two layers were done exactly as in actual CAN system.

The Handel-C program was written without taking advantage of parallelism. The three nodes operate in a sequential manner with time-slicing to give the impression of parallelism. It is possible to write the program so that true parallelism can be achieved. This is discussed in the next chapter.

The controller nodes also operate without a reference clock. However, since each of the nodes takes turn to write and read one bit value at each execution cycle, they appear to operate by referencing to a "virtual clock". However this virtual clock has variable period. Its period increases and decreases cycle to cycle depending upon the calculation and operation it the program has to execute.

Overall, the set of programs gave a satisfactory result in emulating the function of a controller area network. Vital features of CAN listed below were successfully incorporated:

- Bit-sized data transmission
- Bit stuffing while sending
- Bit de-stuffing while receiving
- Non-destructive bit-level arbitration
- CRC computation

The Handel-C program for the FPGA has been written in modular manner and in such way that it can be easily adapted for parallelism. With little modification it can be turned to contain only a single node. It is possible to run several copies of the program on separate FPGA chips and hard-wire them to each other forming a physical network.

6.3 Meta-level Evaluation

When designing and developing this project, the conventional building cycle for typical Handel-C project was followed. First, a software simulation of the network was written in C/C++. This program simulated the functions of three CAN controller nodes communicating with each other through a bit-sized bus. Protocol used for communication was CAN version 2.0A. Using this program, the correctness of algorithms were checked, particularly the creation of dataframe, bit-stuffing, de-stuffing and non-destructive arbitration process.

Next, the software version was ported to Handel-C. Two major issues faced when porting to Handel-C were the fact that Handel-C is very strongly-typed and that it was a hardware synthesis language. Every values used in a Handel-C program must be of fixed widths. If the width of a value was not specified, the compiler will try to compute

it during the build process. However, most of the time, it failed to do so, thus the codes must be changed to be more specific.

Being a hardware synthesis language, there are some constructs in Handel-C that were specifically designed for hardware orientated operations such as bit manipulations, RAM data type etc. Parts of the original C/C++ codes were changed to take advantage of these constructs, particularly those for bit manipulations. For example, instead of multiplying a value by eight (`x * 8`), left-shifts were used instead (`x = x << 3`).

It was also discovered that when the value of a variable of a certain width is increased (x++), it will never overflow. It the value exceeded the maximum allowed for that particular width, it will change to zero instead. For example, the following *for-loop* will never end because the value for variable x will never reach 8:

```
void neverending( void )
{
    unsigned int 3 x;
    unsigned int 8 y;

    for( x=0; x<8; x++)
    {
       y = 0@(3*x);
       x++;
    }
}
```

The Handel-C program can be run in a simulation mode or as loaded into and run on an FPGA. The simulation mode has only simple input output channels to allow for debugging. This has led to some difficulty in verifying vital values generated inside the program. This was subsequently resolved by writing the values to several files and reading them again into a spreadsheet as mentioned earlier.

The DK1 package also came with a few plugins that simulate the function of input/output hardware (seven-segment display and wave simulator) that can be connected to an FPGA. Additional plugins can also be developed for other hardware

simulations. However, due to time constraint, this avenue was not pursued. It could have made the simulation more accurate and realistic.

The final stage of the development was writing a separate host program that communicates with the Handel-C program through library functions. Once again the codes concerning data input/output need to be altered. Debugging the communication process was time-consuming because there were two programs to edit and building the target bit file was a time-consuming two-step process.

An alterative to the above issue would be writing a general purpose host that handles the host-FPGA communication and writing the FPGA program directly in Handel-C. If the host-FPGA communication can executed efficiently, more effort can be put into getting the Handel-C codes right. However, this alternative also could not be pursued due limited resources.

Overall, the methodology employed was suitable for design and development of the system. However, alternatives mentioned above are worth exploring.

6.4 Summary

The system was found to be an accurate representation of a basic CAN operating using standard dataframes. The methodology and techniques employed were found to be suitable for the project. However it is also noted that variations to the techniques employed are worth considering.

7 Conclusion & Further Work

7.1 Conclusion

Functionally, the system developed as the result of this project satisfies the objectives and requirements identified in preceding chapters. It has the fundamental properties of a controller area network and operates as such. The system has been tested and found to comply with the basic specification of CAN. Some of the more critical features of CAN that have been satisfied were:

- Compliance with the OSI layers in CAN: Physical Layer and Datalink Layer; and additionally an application layer as an interface between the controller nodes and user.
- Data transmission between three CAN controller nodes through a virtual CAN bus through CAN Protocol Version 2.0A.
- Creation of Standard CAN Version 2.0A dataframe including CRC computation.
- Transmissions of dataframes into the CAN bus with bit stuffing computation.
- Retrievals of data from the bus and re-creation of dataframes from the data retrieved including CRC.
- Communication between datalink layer (FPGA) and application layer (host).
- Simple user interface and status reports.

The system has been build as a basis for further exploration of a reconfigurable system. It has been designed with future enhancement in mind. Its codes can be easily altered to add more features. These features can be in form of CAN functionalities or

exploration in the aspect of system development studies as discussed in the following section.

7.2 Further Work

Several enhancements can be made to the existing system as discussed in the following sections.

7.2.1 Parallelising

In this implementation, the CAN nodes are not running parallel to each other. They are executed one after another in a time-slicing fashion. Handel-C allows several copies of the same function to be executed in parallel to each other. This is done by defining the function as an array of functions. Without this construct, the only safe way to run a function in parallel with itself would be to explicitly declare two functions with different names.

The syntax for the definition of a function array of an arbitrary size is as follows:

```
returnType Name[Size](parameterList);
```

For example, to redefine the write_controller() function as an array of function, we can re-write it to be as follows:

```
void controller_write[3]( void )
{
      . . .
}
```

In the above example, only three nodes were defined. The number of nodes can be increased simply by increasing the size of the array. Other functions should also be parallelised so there will not be a conflict when two functions from the same array try to access an identical function at the same time. However, care must still be taken because

sometimes the function arrays will try to access the same variable at the same time. The best way of solving this potential conflict is by utilizing the semaphore construct of Handel-C.

Semaphores are declared with the sema keyword. For example, a semaphore for protecting the CAN bus can be defined as:

```
sema busSema;
```

Semaphores that have been defined can be controlled by two functions:

- `trysema(semaphore)` — tests to see if the semaphore is owned. IF not, it returns one and take ownership of the semaphore. If it is, it returns a zero.
- `releasesema(semaphore)` — releases a semaphore that was previously taken by `trysema(semaphore)`.

For example a protected write_bus() function array can be defined as follows:

```
void write_bus[3]( sema *busSema, unsigned int 1 bit)
{
      while( trysema(*busSema)==0) delay;  // wait till bus is free
      bus_value = bus_value && bit;

}
```

7.2.2 Physical Bus

The three CAN controller nodes in the system were interconnected through a bus that complied to the properties of a CAN bus in term of its functionality only. If each of the nodes is placed on a separate FPGA chips, they can be connected to each other physically. The most popular form of CAN bus is a piece of twisted pair cable which can be easily obtained because it is widely used in Ethernet.

With minor modifications, copies of the Handel-C program can be run on its own chip. Each chip can be placed on a customized printed circuit board each with line buffer/driver circuitry to drive the bus at correct voltage and current level. The functions that write to the bus and read from it have to be re-written because they would be writing to and reading from a physical bus.

By adding a clock function in each node, the controllers should be able to communicate with each other correctly. Of course, error handling routines need to be enhanced so that the system can handle problems associated with serial data communications such as synchronization, bit drifting, data losses etc. This would be a very challenging but highly interesting endeavour.

7.2.3 Graphical User Interface

At the user end, ease of use can be enhanced with a graphical user interface (GUI) where the status of operation can be seen clearly, preferably with some graphical representation of the network showing th movement of data from one node to another. It will also be possible to showing the status of each node with colour or iconic indicators. For example: gray when idle, green when sending, blue when arbitrating and red when receiving. This GUI addition can be coded C++ with Microsoft Foundation Class (MFC).

7.2.4 CRC Calculation

This implementation uses a 15 bit polynomial for its CRC computation. In this project, the basic method of CRC calculation has been used. However, it is possible to build a lookup table based on the 15 bit polynomial and write a faster code by utilising the said table. This technique can improve the overall performance of the CAN controllers.

7.3 Summary

Overall, a basic but expandable system been produced out of this project and useful skill and knowledge were acquired through the whole process from the investigation into the subject matter to the writing of this dissertation.

References

1 J G Ganssle, "The Art of Designing Embedded Systems", Newnes 2000

2 T M Conte, "Choosing the Brain(s) of an Embedded System", IEEE Computers July 2002

3 Philips Semiconductors (http://www.semiconductors.philips.com)

4 Handel-C Language Reference Manual, Version 2.1, Celoxica 2001.

5 M Serra & W B Gardner, "Hardware/Software Codesign - introducing an interdisciplinary course", Univ. of Victoria, Victoria, B.C. Canada, WCCCE Conference - Vancouver, 1998.

6 A Jantsch, P Ellervee, J Öberg, A Hemani, H Tenhunen "A Software Oriented Approach to Hardware/Software Codesign", Royal Institute of Technology, Kista, Sweden, 1994

7 W Wolf "Computers as Components: Principles of Embedded Computing System Design", Morgan Kaufmann Publishers, 2001; pp567-578.

8 CAN Specification Version 2.0, Robert Bosch GmbH, 1991

9 KW Tindell, H Hansson, AJ Wellings "Analysing Real-Time Communications: Controller Area Netwrok (CAN)"

10 G Coulouris, J Dollimore, T Kindberg, "Distributed Systems – Conceptual Design", Addison-Wesly 3/ed 2001

11 LB Fredriksson, "A CAN Kingdom Rev. 3.01", CAN Kingdom Specification Manual, KVASER AB 1995

12 Intel Introduction to In-Vehicle Networking, (http://developer.intel.com/design/auto/autolxbk.htm) (April 2002)

13 Philips Semiconductors P8xCE598 8-bit microcontroller with on-chip CAN (http://www.-semiconductors.philips.com)

14 Philips Semiconductors SJA1000 Stand-alone CAN controller (http://www.semiconductors.philips.com)

15 L Fredriksson "Controller Area Network and the CAN protocol for machine control systems", Mechatronics Vol 4 No.2 pp159-192; 1994

16 KM Zuberi, KG Shin, "Real-time decentralised control with CAN", Proc. IEEE Conference on Emerging Technologies and Factory Automation, pp 93-99, Nov 1996

17 L Rauchaupt, "Performance analysis of CAN based systems", 1st International CAN Conference, Mainz 1994

18 BP Upender, A Dean, "Variabliity of CAN Network Performance"

19 L Lemus, J Gracia and P Gil "Designing, modelling and implementing a Controller Area Network (CAN) on a FPGA using VHDL, 1999

20 K Lennartsson and L Fredriksson, "Fundamental parts in SDS, DeviceNet and CAN Kingdom", 1995

21 L Fredriksson "Bluetooth in Automotive applications", Bluetooth '99 Conference, 1999

22 Kvaser CAN Pages — The CAN Protocol, http://www.kvaser.com/can/protocol (April 2002)

23 I+ME ACTIA, "Introduction to CAN – Controller Area Network", http://www.ime-actia.com/-can_intro.htm (1st March 2002)

24 S Nilsson "Controller Area Network – CAN Information" 1997, http://www.algonet.se/-~staffann/developer/CAN.htm (March 2002)

25 Accutest "The CAN Guru Classroom", http://www.accutest.co.uk (April 2002)

26 C Sweeney "Hardware Design Methodologies", Celoxica Limited 2002

27 Celoxica, "RC1000 Software Reference Manual", Celoxica Limited 2001

28 Celoxica, "DK1 Design Suite User Manual", Celoxica Limited 2001

Listing L1 – setup_message() function

```
void setup_message( unsigned int node )
{
      unsigned int i;

      Buffer[0] = node;                    // node number
      Buffer[1] = msgID[node];             // Message ID
      Buffer[2] = datalength[node];        // Number of bytes of data

      for (i=0; i<datalength[node]; i++)
            Buffer[3+i] = data[node][i];   // The data...

}
```

Listing L2 – send_fpga() function

```
void send_fpga( unsigned char msgtype, unsigned char Items )
{
      unsigned char ReturnVal, i;

      PP1000SetGPO(Handle, 1);                    // Set flag

      PP1000WriteControl(Handle, msgtype);
      PP1000ReadStatus(Handle, &ReturnVal);

      if( msgtype==MESSAGEmsg || msgtype==STATUSmsg )
      {
            PP1000WriteControl(Handle, Items);
            PP1000ReadStatus(Handle, &ReturnVal);

            for (i=0; i<Items; i++)
            {
                  PP1000WriteControl(Handle, Buffer[i]);
                  cout << " " << (int) Buffer[i] << Buffer[i];
                  PP1000ReadStatus(Handle, &ReturnVal);
            }
      }

      PP1000SetGPO(Handle, 0);                    // Reset flag
      cout << "\nOK." << endl;
}
```

Listing L3 – CRC calculation routine

```
remainder = 0;
i=0;

while ( i<totlen )
{
      if ( i<msglen )  nxtbit = dataframe[ctrlno][i];

      else  nxtbit = 0;

      bit14 = ( remainder ^ 0b100000000000000 ) ? 1:0; // Check bit
14
      crcnxt = nxtbit ^ bit14;
      remainder = remainder << 1;                  // shift left 1 bit
      remainder = remainder & 0x7FFE;              // clear bit#0

      if (crcnxt) remainder ^= 0x4599;             // remainder XOR poly

      i++;
}
```

Spreadsheet for Data Verification (shown partially)

1	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
2	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
3	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
4	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
5	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
6	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
7	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
8	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
9	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
10	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
11	WAIT	0	0	0	1	0	0	WAIT	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
12	WAIT	0	1	0	1	0	0	WAIT	0	1	0	1	0	0	IDLE	0	0	0	1	0	0	1
13	WAIT	0	2	0	1	0	1	WAIT	0	2	0	1	0	1	IDLE	0	0	0	1	0	0	1
14	WAIT	0	3	0	1	0	2	WAIT	0	3	0	1	0	2	IDLE	0	0	0	1	0	0	1
15	WAIT	0	4	0	1	0	3	WAIT	0	4	0	1	0	3	IDLE	0	0	0	1	0	0	1
16	WAIT	0	5	0	1	0	4	WAIT	0	5	0	1	0	4	IDLE	0	0	0	1	0	0	1
17	WAIT	0	5	0	1	0	0	WAIT	0	5	0	1	0	0	IDLE	0	0	0	1	0	0	1
18	WAIT	0	6	0	1	0	1	WAIT	0	6	0	1	0	1	IDLE	0	0	0	1	0	0	1
19	WAIT	0	7	0	1	0	2	WAIT	0	7	0	1	0	2	IDLE	0	0	0	1	0	0	1
20	WAIT	0	8	0	1	0	3	WAIT	0	8	0	1	0	3	IDLE	0	0	0	1	0	0	1
21	WAIT	0	9	0	1	0	4	WAIT	0	9	0	1	0	4	IDLE	0	0	0	1	0	0	1
22	WRITE	0	0	0	1	0	0	WRITE	0	0	0	1	0	0	IDLE	0	0	0	1	0	0	1
23	WRITE	1	1	0	0	0	0	WRITE	1	1	0	0	0	0	READ	0	1	0	0	0	0	0
24	WRITE	2	2	0	0	1	1	WRITE	2	2	0	0	1	1	READ	0	2	0	0	0	1	0
25	WRITE	3	3	0	0	2	2	WRITE	3	3	0	0	2	2	READ	0	3	0	0	0	2	0
26	WRITE	4	4	0	0	3	3	WRITE	4	4	0	0	3	3	READ	0	4	0	0	0	3	0
27	WRITE	5	5	0	0	4	4	WRITE	5	5	0	0	4	4	READ	0	5	0	0	0	4	0
28	WRITE	5	5	1	1	0	0	WRITE	5	5	1	1	0	0	READ	0	5	0	1	0	0	1
29	WRITE	6	6	0	0	0	0	WRITE	6	6	0	0	0	0	READ	0	6	0	0	0	0	0
30	WRITE	7	7	0	0	1	1	WRITE	7	7	0	0	1	1	READ	0	7	0	0	0	1	0
31	WRITE	8	8	0	0	2	2	WRITE	8	8	0	0	2	2	READ	0	8	0	0	0	2	0
32	WRITE	9	9	0	0	3	3	WRITE	9	9	0	0	3	3	READ	0	9	0	0	0	3	0
33	WRITE	10	10	0	0	4	4	WRITE	10	10	0	0	4	4	READ	0	10	0	0	0	4	0
34	WRITE	10	10	1	1	0	0	WRITE	10	10	1	1	0	0	READ	0	10	0	1	0	0	1
35	WRITE	11	11	0	0	0	0	WAIT	0	11	1	0	1	0	READ	0	11	0	0	0	0	0
36	WRITE	12	12	1	1	0	0	WAIT	0	12	1	1	1	0	READ	0	12	0	1	0	0	1
37	WRITE	13	13	0	0	0	0	WAIT	0	13	1	0	1	0	READ	0	13	0	0	0	0	0
38	WRITE	14	14	0	0	1	1	WAIT	0	14	1	0	1	1	READ	0	14	0	0	0	1	0
39	WRITE	15	15	0	0	2	2	WAIT	0	15	1	0	1	2	READ	0	15	0	0	0	2	0
40	WRITE	16	16	0	0	3	3	WAIT	0	16	1	0	1	3	READ	0	16	0	0	0	3	0
41	WRITE	17	17	1	1	0	0	WAIT	0	17	1	1	1	0	READ	0	17	0	1	0	0	1
42	WRITE	18	18	0	0	0	0	WAIT	0	18	1	0	1	0	READ	0	18	0	0	0	0	0
43	WRITE	19	19	0	0	1	1	WAIT	0	19	1	0	1	1	READ	0	19	0	0	0	1	0
44	WRITE	20	20	1	1	0	0	WAIT	0	20	1	1	1	0	READ	0	20	0	1	0	0	1
45	WRITE	21	21	1	1	1	1	WAIT	0	21	1	1	1	1	READ	0	21	0	1	0	1	1
46	WRITE	22	22	1	1	2	2	WAIT	0	22	1	1	1	2	READ	0	22	0	1	0	2	1
47	WRITE	23	23	1	1	3	3	WAIT	0	23	1	1	1	3	READ	0	23	0	1	0	3	1
48	WRITE	24	24	1	1	4	4	WAIT	0	24	1	1	1	4	READ	0	24	0	1	0	4	1
49	WRITE	24	24	0	0	0	0	WAIT	0	24	1	0	1	0	READ	0	24	0	0	0	0	0
50	WRITE	25	25	1	1	0	0	WAIT	0	25	1	1	1	0	READ	0	25	0	1	0	0	1
51	WRITE	26	26	1	1	1	1	WAIT	0	26	1	1	1	1	READ	0	26	0	1	0	1	1
52	WRITE	27	27	1	1	2	2	WAIT	0	27	1	1	1	2	READ	0	27	0	1	0	2	1
53	WRITE	28	28	1	1	3	3	WAIT	0	28	1	1	1	3	READ	0	28	0	1	0	3	1
54	WRITE	29	29	1	1	4	4	WAIT	0	29	1	1	1	4	READ	0	29	0	1	0	4	1
55	WRITE	29	29	0	0	0	0	WAIT	0	29	1	0	1	0	READ	0	29	0	0	0	0	0
56	WRITE	30	30	1	1	0	0	WAIT	0	30	1	1	1	0	READ	0	30	0	1	0	0	1
57	WRITE	31	31	1	1	1	1	WAIT	0	31	1	1	1	1	READ	0	31	0	1	0	1	1
58	WRITE	32	32	1	1	2	2	WAIT	0	32	1	1	1	2	READ	0	32	0	1	0	2	1
59	WRITE	33	33	1	1	3	3	WAIT	0	33	1	1	1	3	READ	0	33	0	1	0	3	1
60	WRITE	34	34	1	1	4	4	WAIT	0	34	1	1	1	4	READ	0	34	0	1	0	4	1
61	WRITE	34	34	0	0	0	0	WAIT	0	34	1	0	1	0	READ	0	34	0	0	0	0	0
62	WRITE	35	35	1	1	0	0	WAIT	0	35	1	1	1	0	READ	0	35	0	1	0	0	1
63	WRITE	36	36	0	0	0	0	WAIT	0	36	1	0	1	0	READ	0	36	0	0	0	0	0
64	WRITE	37	37	0	0	1	1	WAIT	0	37	1	0	1	1	READ	0	37	0	0	0	1	0
65	WRITE	38	38	1	1	0	0	WAIT	0	38	1	1	1	0	READ	0	38	0	1	0	0	1