Supporting Transparent Dataflow Messaging in Distributed Power Electronics Control Systems

Parool Mody

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science

Dr. Stephen Edwards, Chairman
Dr. Dushan Boroyevich
Dr. Eunice Santos

15 December 2003
Blacksburg, Virginia

Keywords: Inter-processor communication, RPC, message passing, dataflow, asynchronous messaging, embedded systems

Supporting Transparent Distributed Messaging for Dataflow Applications in Power Electronics Control Systems

Parool Mody

(ABSTRACT)

This thesis presents the design and implementation of a transparent messaging protocol for distributed communication between processors. The processors are designed using dataflow architecture. The protocol ensures transparent asynchronous communication between distributed processes. The protocol is designed such that an application can run without change in virtually any kind of distributed configuration, where configuration is the number of controllers used in the system plus the processor allocation strategy used. It also enables an automated processor allocation strategy to transparently configure an application for any number of processor nodes without requiring any changes or recompilation. The protocol works well even for single-controller applications and for a pre-defined allocation of processors to controllers. The thesis further includes an analysis of the time required for one complete cycle of inter-processor communication.

This work was supported primarily by the Office of Naval Research under Award Number N00015-01-1-0954 and the ERC Program of the National Science Foundation under Award Number EEC-9731677.

Acknowledgements

I would like to thank my advisor Dr. Stephen Edwards for his able guidance during each stage of this research. His encouragement was a great source of motivation for me. I am also grateful to Dr. Eunice Santos and Dr. Dushan Boroyevich for their valuable suggestions.

I am grateful to all the members of the entire PEBB group for their support during my thesis. I would like to thank Kuljeet Singh and Jinghong Guo for helping me get acquainted with the project. I am also thankful to Jerry Francis, for his help in testing my protocol on the hardware.

And finally, I would like to express my heartfelt gratitude to family and friends for their support and encouragement, without which this thesis wouldn't have been possible.

Parool Mody

List of Figures

Chapter 1

Introduction

Applications designed for single processor systems often require many code changes to enable their use in multi-processor systems. Code changes are also required whenever the system is reconfigured. This thesis presents a messaging protocol for communication between processes on different processors, designed for use with dataflow architecture applications. Dataflow architecture is a component-based architecture that automatically supports reusability in embedded systems. The messaging protocol is independent of the number of processors and of the assignment of processes to processors. Hence, it requires no changes in code when the system is reconfigured. Also, the same application can be used without any change on both single processor and multi-processor systems.

We have implemented our protocol for use in power electronics control systems designed using dataflow architecture. The processors are placed on a dual ring fiber optic local area network and communicate using a communication protocol called Dual Ring PESNet (DRPESNet) [Francis02].

1.1 System Architecture

Modularity in power electronic systems is achieved by separating the control algorithms from device specific implementations. The design architecture for plug and play power electronics building blocks consists of application managers and hardware managers, connected in a daisy chained dual ring using 125 Mbps optical fiber. The application manager is designed to control through software, any kind of power electronics hardware. The hardware managers handle all the hardware specific tasks.

The application managers and hardware managers communicate through a protocol called DRPESNet [Francis02]. DRPESNet is a master-slave communication protocol, which supports dynamic node address assignment, network configuration, fault tolerance and basic network communication. The protocol also allows applications to have more than one master processor.

Communication on the ring is in the form of packets. The ring will have multiple packet slots. Any of the master processors can initiate communication on the ring by replacing an empty packet slot with a data packet. The size of the packet is fixed and is specified by the application designer at configuration time.

The software for plug and play power electronics building blocks has been designed using dataflow architecture. Dataflow architecture [Guo02, Singh02-1, Singh02-2] is a data-driven software architecture, which supports reusability, scalability and maintainability. It consists of data triggered independently

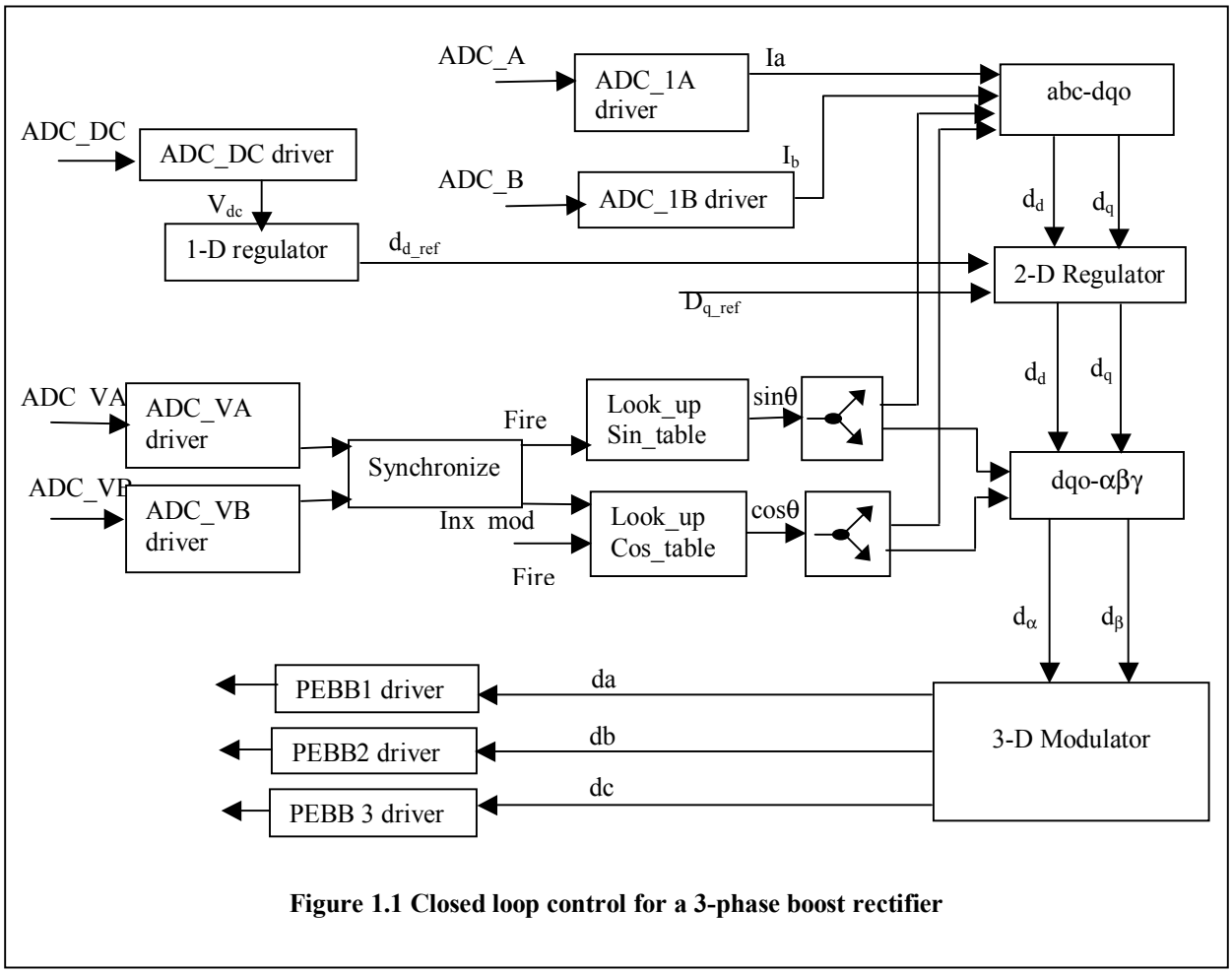


Figure 1.1 Closed loop control for a 3-phase boost rectifier

executing entities called the elementary control objects (ECOs), interconnected by buffered message pipes called data channels. The ECOs execute in parallel and communicate with each other through data channels. Figure 1.1 shows a closed loop control for a 3-phase boost rectifier [Singh02-2]. The rectangular boxes represent the ECOs while the arrows represent the data channels connecting the ECOs.

Data channels are unidirectional i.e. data flows only from source ECOs to sink ECOs. Data type information is provided for each data channel to indicate the type of data that will be carried by the data channel. This enables detection of certain kinds of interconnection errors early on during development rather than later during operational testing. Data channels are implemented as circular buffers to enable communication between ECOs operating at different speeds. This provides for asynchronous communication between the ECOs.

The ECOs are implemented as threads. Threads are capable of executing concurrently and have their own run time stack. The ECOs are not aware of other ECOs in an application. They simply read data from incoming data channels, perform computation and write data to outgoing data

channels. As ECOs are independent entities, they are able to execute concurrently. A control application can thus be built by dividing the control algorithm into ECOs, picking up the required ECOs from the library and connecting the ECOs together using data channels into a desired pattern.

ECOs are scheduled using firing rules. A firing rule consists of a firing mask and a priority. The firing mask holds a binary number, which indicates data channels that should contain data in order to trigger the firing mask. For e.g. a firing mask with value 00000101 gets triggered if incoming data channels 1 and 3 contain data. Priority indicates the new priority that will be assigned to the ECO after the associated firing mask is triggered. An ECO can have more than one firing rule associated with it.

An ECO gets blocked if it tries to read from an empty data channel. An ECO attempting to write to a full data channel may get blocked, overwrite the newest data element or overwrite the oldest data element in the data channel. The action taken depends on the value of the overflow_style field of the data channel. A read from a data channel can unblock an ECO waiting to write data into the data channel. A write to a data channel can fire the sink ECO waiting for input on that data channel. The ECOs communicate with each other only through APIs. Thus any change in an ECO does not require change in other ECOs.

A **D**ataflow **A**rchitecture **R**eal-time **K**ernel (DARK) [Singh02-1, Singh02-2] operating system is created to support ECO based dataflow applications. DARK is responsible for scheduling the ECOs based on their priorities, reading and writing data into data channels, firing of ECOs, providing support for device drivers and handling of interrupts. Initializing the system at startup using the data flow graph also forms the responsibility of DARK.

For applications with only one master, all the processes are allocated to that single application manager. In case of applications with more than one master, the allocation of processes (ECOs) to the processors (application managers) can be done by manual or automated process allocation strategy.

1.2 Problem Statement

Multi-processor power electronics systems have distinct advantages over single processor systems. A multi-processor system has more processing power and hence can be used to perform a larger number of tasks in a given period of time compared to a single processor system. Also, it is often cheaper to interconnect several smaller processors instead of using a single big processor. We can make use of a number of standard components available in the market and connect them together to develop the desired application. However applications developed in this way require distribution and location transparency in order to achieve such advantages. By location transparency, we mean that the components should be able to interact with each other without being aware of their location. Distribution transparency is achieved when an application designed for a single processor system can be distributed without requiring any change in the application code.

However, in most multi-processor systems, an application needs to be modified when there is a change in the number of processors in the system or when there is a change in the allocation of processes to the processors. This is because in most systems, communication between processes is achieved by hard coding the names of sender and receiver processes. Thus, any change in the system configuration requires recompilation and relinking of tasks. In case of systems, which do not use hard coded sender and receiver names, a lot of overhead is involved in linking the ports. Also, most systems require specific constructs to be added to an application code to enable its usage in a distributed environment.

This thesis focuses on the problem of *how we can design a messaging protocol to ensure transparent communication between processes on different processors, designed using dataflow architecture.* To ensure transparent inter-processor communication for dataflow-based applications, the protocol must satisfy the following criteria:

- Asynchronous Communication between dataflow components: In dataflow architecture, the intra-processor communication is done asynchronously through data channels. Hence, in order to provide transparent behavior, asynchronous communication should be ensured even for ECOs on different processors.

- Location Transparency: Location transparency helps make the code independent of the location of the processes. This enables dynamic allocation of processes to processors without requiring any change in the application code.

- Distribution Transparency: Single processor applications often need to be ported to multi-processor systems for greater processing power. In order to achieve this without any code change, it is required that the application code written for communication between processes on the same processor work without any change for communication between processes on different processors.

- Efficiency: Multiprocessor systems offer lot of advantages over systems with a single processor. They have more processing power and allow for processes to be executed in parallel. However, they also include a lot of overhead due to inter-processor communication. In order to reduce the overhead and to take maximum benefits out of multiprocessor systems, the communication between different processors must be efficient.

- Fault tolerance: In order to ensure transparent inter-processor communication, it is required that the data be received by the receiver ECO in the same order as it is sent by the sender ECO. This transparency should be maintained even if there is a node or a link failure in the network.

The protocol guarantees both location and distribution transparency. The ECOs within an application communicate with each other without being aware of their locations. The number of processors has no bearing on the way an application is implemented. The protocol will work fine for applications with variable number of processors and / or process allocation strategy without requiring any code change. The protocol also provides fault tolerance and ensures orderly arrival

of packets. An additional advantage of this protocol design is that an application with an automated process allocation strategy can reconfigure itself for any number of processor nodes without requiring any code change.

Although this protocol is currently being implemented for power electronics applications, it is designed in such a way so as to allow its implementation on any application designed using dataflow architecture. The protocol is also independent of the underlying communication protocol. It is designed such that it can work with any communication protocol that offers fault tolerance.

The thesis also includes a theoretical analysis of the network parameters to determine the system performance. The analysis uses parameters such as saturation of the network, network speed and size of the packet to determine the time it takes for a complete cycle of the packet and the number of packets that can be exchanged in a single switching cycle. The theoretical analysis can be easily validated by simulating the protocol on the processor controller boards and measuring the time required for one cycle of the protocol. The validation is included as future work of the thesis.

1.3 Overview of the Solution Strategy

ECOs are functionally self-contained entities executing concurrently with each other. An ECO reads data from input data channels; performs computations and writes result onto output data channels. There are no explicit calls between ECOs. Thus, the ECOs are not aware of other ECOs in the system or the location of ECOs with which they communicate. This makes it possible to assign ECOs to different processors and execute them in parallel without making any change to the application. With proper allocation of ECOs to processors, the communication between ECOs can be reduced to the minimum thereby improving performance.

ECOs on a single processor communicate with each other through data channels. In case of ECOs on different processors, the communication is through data channels, which are distributed in nature. Distributed data channels have the same structure as normal data channels. The ECOs are not aware of the distributed nature of the data channel. This allows the ECOs to read or write data from the data channel irrespective of the nature of the data channel and hence of the location of the communicating ECO.

Figure 1.2 shows the transfer of data in the form of a packet from source ECO to destination ECO. When an ECO writes data to a distributed data channel, the operating system packs a copy of the data from the data channel into packet and then sends the packet over the network to the destination ECO through the FPGA. Note that the operating system only sends a copy of the data while the original data is still stored in the data channel. The operating system at the receiver side will unpack the packet and write data into the data channel, identified by the packet.

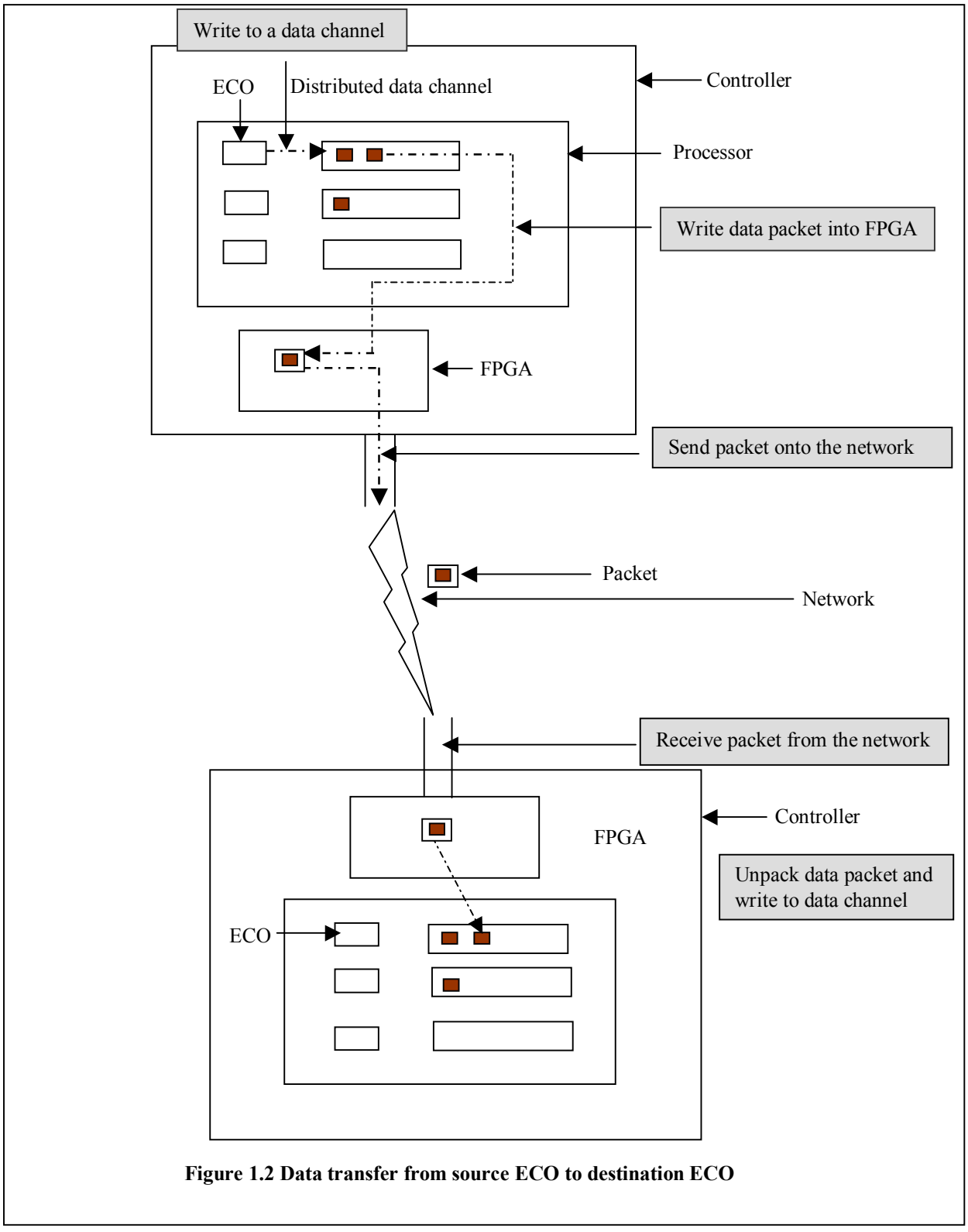


Figure 1.2 Data transfer from source ECO to destination ECO

The Field Programmable Gate Array or FPGA is a type of a hardware chip consisting of large number of logic gates with the interconnection between the gates user programmable. The FPGA is concerned with the monitoring of the network traffic and sending and receiving packets from the network.

In order to ensure reliable delivery of packets, the protocol requires acknowledgement for every data packet sent. Also at a time, corresponding to a single data channel, only one packet can be sent over the network and the next packet is sent only after an acknowledgement is received for the first packet. This ensures that packets are always delivered in order. Also, as the data from the data channel gets deleted only after an acknowledgement is received for that data, all data gets delivered from source data channel to destination data channel. This ensures that even if data packets are lost over the ring, no data will be lost.

If there is not enough space for the entire packet in the data channel on the receiver side, it is possible that only part of the data gets written into the data channel and hence only part of the data gets acknowledged. In that case, only those data bytes that are acknowledged are deleted from the source data channel and an attempt is made to re-send the unacknowledged data bytes.

The protocol is implemented over DRPESNet communication protocol, which ensures that no messages are lost even when there is a node or a link failure.

To summarize, the research contribution of this thesis are:

- Design and Implementation of a transparent messaging protocol
- Theoretical analysis of the protocol performance

1.5 Organization of Thesis

We present the organization of the thesis. The next chapter will provide an overview of dataflow architecture and DRPESNet communication protocol. The chapter will also include description of various inter-process communication protocols and their comparison with our protocol. Chapter 3 will contain a description of the protocol design and the data structures used. In chapter 4, we will provide a complete implementation of the protocol and a description of the entire code in detail. In chapter 5, we will present a theoretical analysis of the protocol performance using network parameters such as speed, network saturation and number of nodes in the network. The chapter will also describe the steps to be taken to validate the theoretical analysis. Chapter 6 will conclude the thesis after giving a brief overview of the work to be done in future. The complete source code will be provided in Appendix 1.

Chapter 2

Background Research

A clear understanding of the system architecture is necessary in order to understand the objectives, design and implementation of the messaging protocol. A description of the system architecture is presented in this chapter followed by a discussion of the protocol objectives that we aim to achieve. The chapter also presents research work done till date to achieve transparency in inter-processor communication. A comparative study of the research done so far will help understand the need for a new messaging protocol for transparent inter-processor communication on dataflow applications.

2.1 System Architecture

The power electronics control system consists of application managers and hardware managers connected in a daisy-chained dual ring and communicating using a communication protocol called DRPESNet [Francis02]. The application managers are concerned with the computation of the control algorithm, the result of which is sent to the hardware managers. The task of the hardware managers is to convert the control information received from the application managers into control commands and fast protection. The software for the application manager is designed using dataflow architecture. To ensure proper execution and communication between the components of dataflow architecture, Dataflow Architecture Real-time Kernel (DARK) operating system has been designed.

Section 2.1 presents an introduction to the universal controller designed for power electronics applications. The Section also includes a brief overview of the DRPESNet communication protocol, dataflow architecture and the DARK operating system. Although the protocol is implemented for communication between universal controllers designed using dataflow architecture with DARK as the operating system and communicating through DRPESNet protocol, the protocol design is independent of the universal controllers, the operating system running on them and of the underlying communication protocol.

2.1.1 Universal Controller

The universal controller [Francis01] designed for plug and play power electronics building blocks can be configured to work in single processor mode, multi-processor mode using shared memory and resources and multi-processor mode using independent memory. The controller has 2 buses – one to support clustered multi-processor systems and the other to interface peripherals and other upper level communication interfaces. The FPGA is implemented as a bridge between the 2 buses.

The communication protocol on the fiber optic ring is PESNet [Francis02]. The controller contains two transceivers as interfaces to the dual fiber optic ring. The transceivers contain both a transmitter and a receiver for sending and receiving data from the ring and are connected to the controller through the FPGA. The FPGA presents these transceivers as memory locations to the DSP. The DSP used on the controller is ADSP-21160. ADSP-21160 has link ports that connect multiple controllers together to create a multi-processor system with independent memory.

The controller architecture is as shown in Figure 2.1. The controller provides several interfaces including PMC interface, synchronous serial port interface (SPI), peripheral expansion header interface to connect analog and digital boards directly to the universal controller and upper level control interface to support different upper level communication interfaces such as motor drives and digital and analog I/O. PMC interface is a PCI based mezzanine interface that enables the universal controller to communicate with the PC during runtime, perform hardware emulation and runtime monitoring of the fiber-optic rings and the 2 buses. SPI enables the DSP to communicate with other processors, universal controllers and peripheral devices using synchronous serial communication. The controller also contains EEPROM, a hex display and an analog to digital converter.

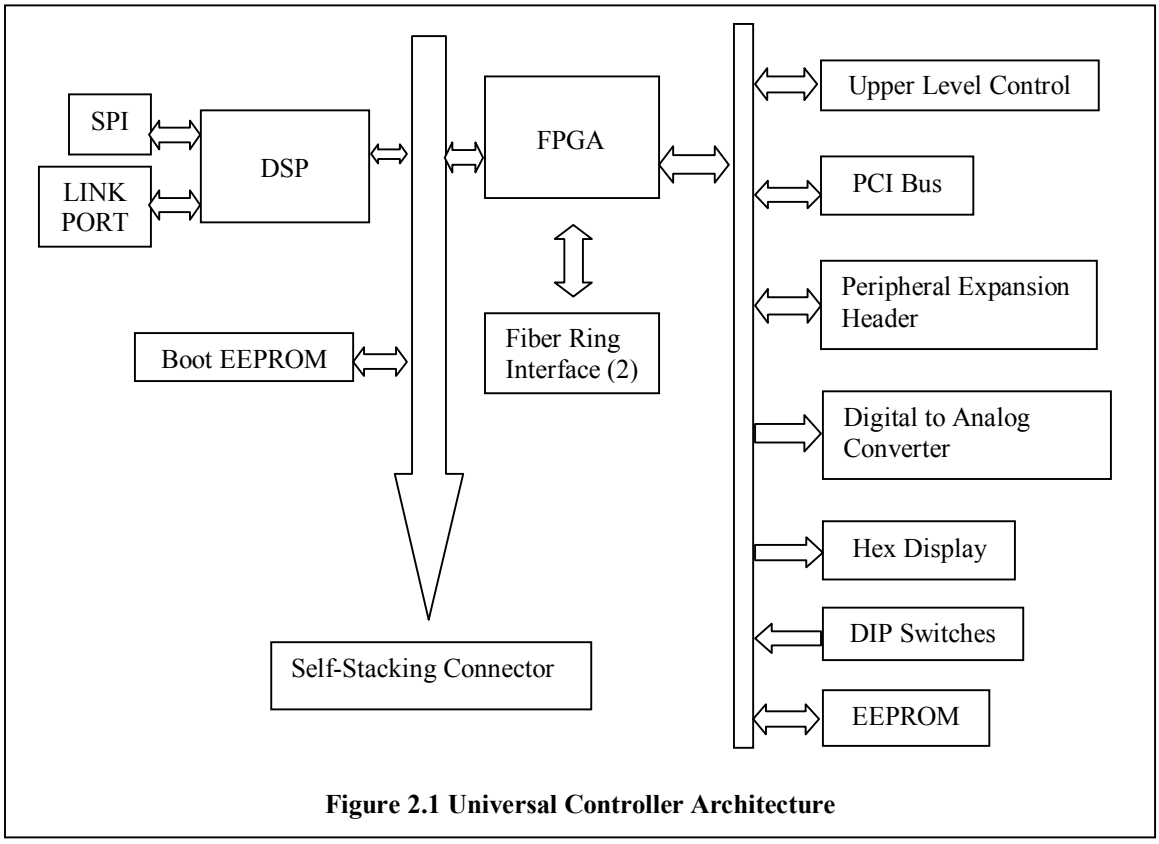


Figure 2.1 Universal Controller Architecture

2.1.2 Dual Ring PESNet Communication Protocol

Dual ring PESNet or DRPESNet [Francis02] is a communication protocol consisting of multiple data slots on a daisy-chained dual ring. The ring is a fiber optic ring with network speed of 125Mbps. The two rings have data flowing in opposite directions to add fault tolerance capability to the protocol. The protocol is capable of supporting applications with more than one master processor node.

DRPESNet operates in 3 modes – normal operation mode, configuration mode and failed mode. Initially when the network powers up, the system is in configuration mode. In the configuration mode, addresses are assigned to the nodes on the ring and the system parameters are decided. Data flows only in the primary ring during configuration and normal operation mode. During normal operation mode, communication can be initiated by any of the master nodes by replacing a NULL_PACKET on the ring with a data packet. A data packet during that mode takes N hops to travel around the ring, where N is the number of nodes in the network.

When there is a node or a link failure, the network enters a failed mode. During failed mode, each node on the ring forwards packets not destined to it on the secondary ring. When a node receives packet with the source address as its own address, it removes the packet from the ring. The secondary ring will be used only for routing and receiving requests. The primary ring will be used for transmission purposes. This will allow the network to return soon to the normal operation mode. It is the task of the master nodes to determine the failed node. Once determined, no data will be sent to the failed node. During the failed mode, a packet takes 2N hops to travel around the ring.

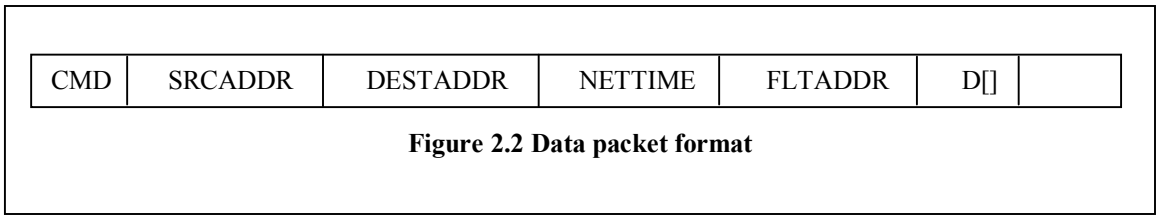


Figure 2.2 Data packet format

The format of the data packet on the ring is as shown in Figure 2.2. The first field is a byte-size CMD field to indicate the packet type. The second and third fields give the address of the source and destination nodes respectively. NETTIME field contains the network time for synchronization purposes. FLTADDR contains the address of the faulted node when the network is in fault tolerance mode. The D[] field contains data. The size of the data field will always be a multiple of 2. The CRC field is used for error checking. The sizes of the address, network time and data are based on the values of the network parameters Pa, Pn and Pd respectively. These values are negotiated based on the application requirements.

2.1.3 Dataflow Architecture

Dataflow architecture [Guo02, Singh02-1, Singh02-2] is data-centric software architecture, consisting of concurrently executing dataflow processes. The dataflow processes are functionally self-contained and independent entities. They communicate with each other through message queues called data channels. The dataflow processes simply read data from their input data channels perform computations and if needed, generate data on their output channels. They do not know anything about the processes with which they communicate or about other processes in an application. Thus, changes made to one process do not require changes to be made to other processes. This makes dataflow-based applications easy to reuse. Also, by identifying common processing tasks, it is possible to build a library of dataflow processes. This will enable rapid development of new applications by using the processes from the library and then wiring them into a desired pattern.

Dataflow applications support reconfigurability. A dataflow application can be described as a graph. The nodes in the graph represent dataflow processes while the arcs represent data channels connecting those processes. As the dataflow processes are loosely coupled, it is very easy to add or remove a node or an arc from an application. This makes dataflow applications easy to reconfigure.

For power electronics system, we refer to dataflow processes as Elementary Control Objects (ECOs) [Guo02]. An ECO gets fired based on data in the input data channels. When fired, the ECO performs some computations and writes data into output data channels, thereby firing other ECOs.

2.1.4 DARK

DARK [Singh02-1, Singh02-2] is a lightweight RTOS designed for power electronics control software developed using dataflow architecture. DARK is concerned with the initialization of the system at startup, scheduling of the ECOs and communication between them, handling of device drivers and external interrupts and allocation of system resources. The initialization of the system is done using the DFG (dataflow graph) descriptor file, which is provided by the application programmer. The file contains information about the ECOs, the data channels connecting them and the implementation of the ECOs in C.

ECOs in DARK are implemented as threads. The threads execute independently and have their own run time stack. A thread can be in one of the seven states at any point in time: nascent, ready, blocked, wait_for_fire, timed_wait, timed_wait_for_fire and dead. The states of the ECO can be described using figure 2.3 [Singh02-1, Singh02-2]. A thread is in a nascent state when the system starts. When the thread is fired, it enters into a ready state. If a thread tries to read from an empty channel or write to a full channel, it gets blocked. Once fired, the ECO can call the wait_for_fire function in order to get fired again. The execution of the ECO can be delayed using a pre-defined time interval using timed_wait function. timed_wait_for_fire is a combination of timed_wait and wait_for_fire functions. Once an ECO completes execution, it is said to be in dead state.

Data channels are implemented as circular buffers to enable communication between ECOs operating at different speeds. The data channels connecting the ECOs are unidirectional that is data flows only from source ECOs to sink ECOs. Data type information is provided for each data channel to indicate the data type that will be carried by the data channel. This enables certain kinds of interconnection errors to be detected early at development time.

Scheduling in DARK is done using firing rules. Firing rule for an ECO defines the data channels that should contain data in order to fire the ECO and the priority of the ECO after it is fired. An ECO can be fired based on different input conditions and thus, can have more than one firing rule. When an ECO is fired, the DARK scheduler places the ECO in the ready queue based on its priority. An ECO can be fired by a read/write operation.

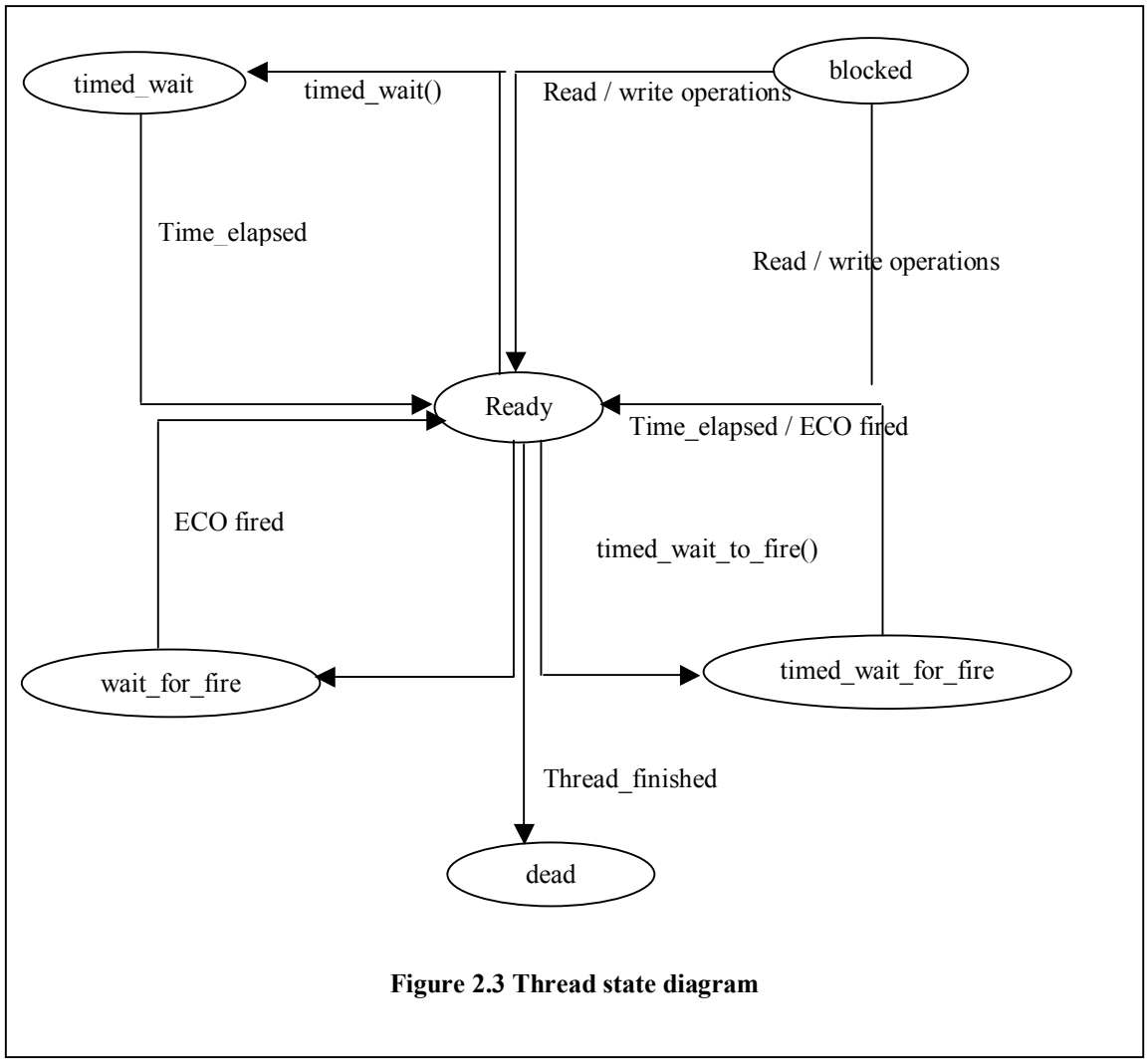


Figure 2.3 Thread state diagram

A read_DC() operation on a data channel can unblock an ECO waiting to write data into that channel. A write_DC() operation can fire an ECO. Thus, the DARK scheduler gets called after read_DC() and write_DC() operations. The DARK scheduler is also called after the execution of timed_wait_for_fire, timed_wait and wait_for_fire functions.

There are four different version of the DARK kernel. They are:

1) Full-featured DARK

2) Non-preemptive DARK

3) Dynamically scheduled single-threaded DARK

4) Statically scheduled single-threaded DARK.

The "full-featured" DARK version allows for dynamic scheduling of the processes based on their priorities and firing rules. After every read and write operation, the DARK scheduler is called which checks to see if any process of equal or higher priority as compared to the priority of the current process is ready for execution. If there is, then the context information of the current process is saved and the next process with equal or higher priority as the current process is set ready for execution.

For non-preemptive dynamically scheduling, the DARK scheduler is invoked only when the current process runs to completion. Dynamically scheduled and statically scheduled are single threaded versions of DARK. Dynamically scheduled single-threaded DARK schedules processes according to their priorities and firing rules. Statically scheduled DARK uses pre-computed order for firing the threads.

Although most of the data channels in DARK support asynchronous communication, DARK also supports synchronous communication using mailboxes. Mailboxes are data channels, which can contain only one data element at a time.

2.2 Interprocessor Communication needs for Dataflow Architecture

In order to keep dataflow applications independent of the exact nature of communication whether inter-processor or intra-processor, inter-processor communication should closely model communication between processes on the same processor. We have listed 5 objectives for inter-processor communication in dataflow applications. They are:

- Asynchronous communication between dataflow components,
- Location transparency,
- Distribution transparency,
- Efficiency and
- Fault tolerance.

In chapter 1, we presented a brief discussion of these objectives. This chapter presents a more detailed description of the objectives of the inter-processor communication for dataflow applications.

Asynchronous Communication between dataflow components: In case of asynchronous communication, the sender and the receiver are not required to be in sync with each other for the communication to take place. The sender and receiver execute concurrently and communicate with each other either through shared data space or through one-way message passing. This kind of communication is often seen in parallel multiprocessor systems where processes on different processors execute concurrently and exchange information through one-way message passing.

Synchronous communication between sender and receiver requires them to be in sync with each other. This kind of communication is often seen in client-server systems where the client gets blocked waiting for the server to process the data and return the result, which the client uses for further processing. In this case, the communication occurs both ways—from client to server and from server back to client.

In dataflow architecture, the components are functionally independent, concurrently executing entities. The sender simply passes data to the receiver and does not care how the information is processed and what result is generated. Hence asynchronous communication is more suited to dataflow architectures.

Location Transparency: Communication is considered to be location transparent if the program elements do not know the location of other program elements with which they are communicating. The application does not contain any location-specific information. Thus, any change in the system configuration does not require the application code to be changed.

In single-processor dataflow applications, communication between components occurs without the components being aware of each other's locations. This location transparency must be maintained even for communication between components on different processors.

Distribution Transparency: Distribution transparency is achieved when an application designed for a single processor system can be used in a multiprocessor system without any code change. This enables an application to be executed without any code change on single as well as multiprocessor systems.

As dataflow-based applications consist of functionally independent program units, they are suited for both single and multiprocessor systems. Distribution transparency ensures that the same application code can be used without any code change on both types of systems.

Efficiency: Since intra-processor communication is through local memory, it takes less time than inter-processor communication where information is to be transferred between processes on different processors. As the application is not aware of the exact nature of communication, the time required for inter-processor communication should be small enough to allow for normal working of the application. In most power-electronics applications, communication between two given processes occurs once every switching cycle. Hence, the messaging protocol must ensure

that the time required for communication between processes on different processors is less than one switching cycle.

Fault tolerance: Fault tolerance is ensuring normal working of the application, even if there is failure of components such a node or link failure. The underlying communication protocol DRPESNet ensures no loss of packets even if there is node or link failure. However, it does not guarantee that all packets sent by a process on the source processor will be received in the correct order by the process at the destination processor. Also, a packet received correctly by the destination processor may not result in transfer of packet data to the sink process due to lack of space in the data channel.

2.3 Transparent Interprocessor Communication Mechanisms

When there are multiple processors working on a single problem, they need to communicate with each other to exchange data and/or results. Communication between processors in a multiprocessor system is called Interprocessor communication or IPC. Lot of work has been done to make IPC transparent to the user, allowing the user to develop applications without being concerned about the communication details. Transparent IPC is important for a lot of reasons. Often the number of processors or the location of programs and objects on those processors is not known at development time. Applications developed for single processor system may need to be run on multiprocessor systems to improve efficiency and scalability. If the application code includes communication information and there is change in system configuration, significant changes will be required to the application to adapt it to the new configuration.

We have classified the research done to achieve transparent IPC into three categories. They are Distributed Shared Memory, Remote Procedure Calls and Other Approaches

2.3.1 Distributed Shared Memory

Distributed shared memory system (DSM) [Protic98] combines the advantages of shared memory and distributed memory systems. It provides shared memory model using physically distributed memory. DSMs can be implemented using hardware and/or software. With the DSM, the application is not required to include code to access data on remote processors. The DSM can normally be classified as fine-grained or course-grained based on their access patterns. DSMs, which allow memory access at the level of a page at a time, are called course-grained DSM while the DSMs that allow access at the level of 128 bytes is considered as fine-grained DSMs.

DSMs implemented in hardware are more like private caches in multiprocessor systems and hence are able to support data access at an object or a block level. They provide greater efficiency as compared to those implemented in software. However, they suffer from lack of flexibility. DSMs implemented in software are more flexible but are generally less efficient than those implemented in hardware. They are implemented as an additional layer above message passing to hide the location specific details from the application. They support data access at a page level. Page level data access helps to take advantage of locality of reference in certain applications. At the same time, decreased granularity causes contention between processors

trying to access unrelated blocks of data in a single page. This is termed as false sharing. This can cause frequent exchange of a single page between processors degrading system performance. This phenomenon is called thrashing [Protic98].

The distribution of data in DSM can be done in 2 ways—replication and migration. In replication, a single copy of data gets replicated over number of processors. In migration, the data is migrated to the accessing processor. The advantages of using DSM as a means of inter-processor communication are as follows:

1) The application need not include any code to support remote code or data access.

2) They provide ease of programming.

3) They provide portability and scalability.

4) DSMs are cost effective compared to shared memory systems.

The disadvantages of using DSM are as follows:

1) In DSMs supporting multiple copies of the data, a protocol needs to be implemented to ensure data consistency across all processors.

2) Ensuring data consistency introduces additional system overhead.

3) Software implementation of DSMs introduces an additional layer above message passing and hence results in additional overhead.

4) Coarse-grained access pattern can result in false sharing and thrashing.

Jackal [Veldema01] is a compiler-supported, fine-grained distributed shared memory system, implemented in Java. Jackal's compiler and run-time system (RTS) allows java programs to be run on distributed systems without any modifications. Jackal allows for a single region to be accessed at a time, where a region contains a java object or part of a java array. All the regions are stored in a single virtual address space, with each region occupying a single virtual-address space across all processors.

[Hu03] provides a handle-based implementation to support efficient and transparent data sharing using both coarse- and fine-grained access patterns. In this system, an object handle is associated with every object and all references to an object are redirected through its handle. Any change in the location of the object requires only the object handle to be change. Thus, a language in which no pointer arithmetic is allowed can be transparently compiled into a handle-based system. A handle table contains all the object handles. Object Identifier (OID) uniquely identifies an object and also serves as an index into the handle table. The Object Identifier is unique across the entire system. An object need not be at the same virtual-address space at all processors. As long as the entry in the handle table corresponding to an object points to a proper location, an object can be allocated at different virtual addresses on different processors.

2.3.2 Remote Procedure Calls

Procedure calls have been widely used for communication between different program elements in a single processor. This gave rise to the concept of a remote procedure call to transfer data and control information across distributed processors. In a remote procedure call, the remote procedure name and parameters are passed to a remote processor where the desired procedure gets executed and the results returned back to the invoking procedure. When a remote procedure is invoked, the calling procedure gets blocked and resumes control only when it receives the result. From the application point of view, it appears as though the procedure were executed on the same processor.

We will be discussing three most widely used mechanisms for RPC.

2.3.2.1 Common Object Request Broker Architecture (CORBA)

CORBA [OMG99] is a distributed object model and serves to execute remote procedure calls. The main component of CORBA is an Object Request Broker core, also called as ORB core, which uses the General Inter-ORB protocol (GIOP) to communicate with objects across the network [Bacellar98].

The CORBA architecture is described using Figure 2.4 [Bacellar98]. In order to create an object implementation on the remote side, the client object calls the sub. The sub obtains the address of the remote object from the Interface repository and passes the request to the client side ORB (cORB). The cORB passes the request to server side ORB (sORB). The sORB loads the server program into the memory. The Object Adapter creates the skeleton object and the object implementation for the server program and passes the reference to the object implementation to the sORB. The object reference is sent back to the cORB and from there back to the stub. The stub returns the reference to the client object.

In order to invoke the object implementation, the client calls the stub. The stub marshals the parameters and passes the request to the cORB. The cORB sends the request to the sORB. The object adapter receives the request and passes it to the skeleton object. The skeleton object unmarshals the parameters and then calls the method on the object implementation. The result generated is then passed back to the client.

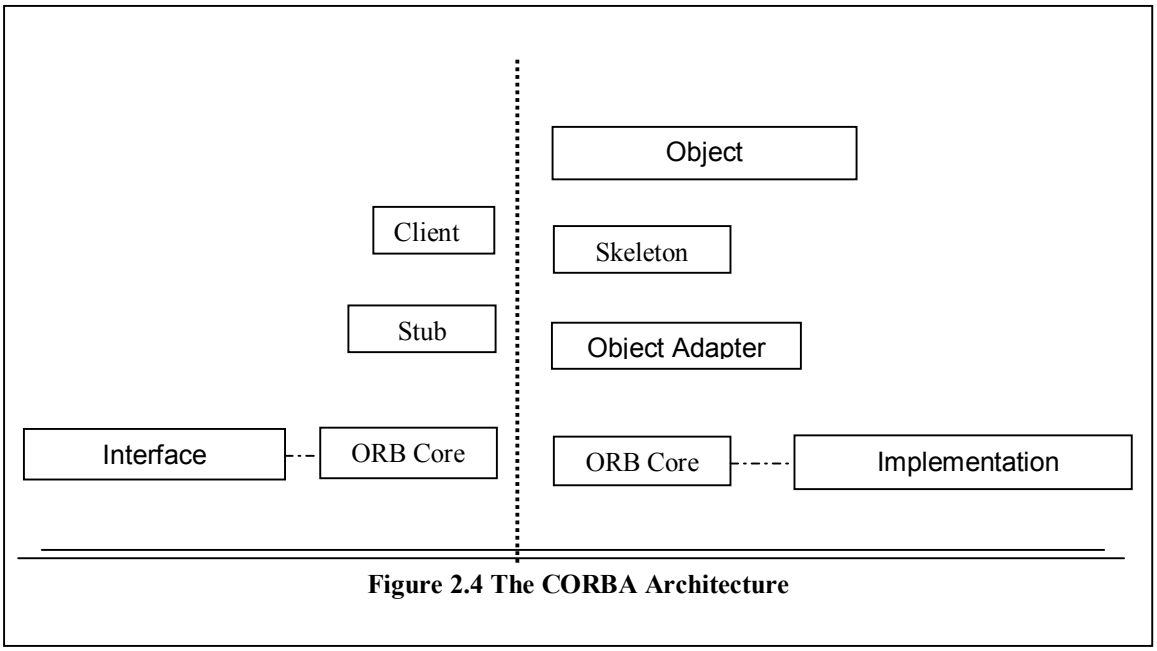


Figure 2.4 The CORBA Architecture

CORBA supports three invocation models—one-way operations, synchronous two-way and deferred synchronous operations using the dynamic method invocation (DII) [Schmidt98]. [Arulanthu00] describes two models to support asynchronous messaging in CORBA. In the first model called the Polling model, the client can poll the value of a poller class variable to determine if the result of a remote procedure call has arrived. In the second model called the Callback model, the client passes an object reference of a reply handler to the client ORB. The client ORB stores the handler and invokes the appropriate call back operation on the handler, when it receives reply from the server.

Disadvantages:

1) CORBA is not completely distribution transparent. Client and sever applications do need to include CORBA-specific calls.

2) Services like dynamic invocation interface, interoperability extensions and language neutral features result in lot of additional overhead. These services require a lot of memory and processing cycles and place a heavy burden on embedded applications where there is a scarcity of these resources [Bacellar98].

3) CORBA is not very efficient when used to invoke objects on the same processor. Hence in order to obtain efficient intra-processor communication, specific location of the program elements must be known before compile-time.

4) Although CORBA provides asynchronous messaging, it still requires the client to periodically poll the poller class in polling model. In case of Callback model, the client developers have to decide on how to connect the reply with the original request that is whether to use some kind of request id to distinguish between requests and whether to

use a separate reply handler for each request. Also, the client must be prepared to handle "inversion of control" by using callback to handle the incoming reply [Arulanthu00].

2.3.2.2 JAVA Remote Method Invocation (JRMI)

JRMI is a distributed object model from SUN Microsystems [Sun96] [Sun99]. It extends Java to support remote method invocation. JRMI requires all client and server interfaces to be written in Java. The JRMI architecture consists of three layers [Ahuja00].

1) The stub/skeleton layer: consists of stubs on the client side and skeleton on the server side. These serve as interfaces for client and server objects to interact with each other.

2) The remote reference layer: serves as the middleware between the stub/skeleton layer and transport protocol layer.

3) The transport protocol layer: sets up and manages the connection to remote objects and sends remote object requests across the network.

The client uses the stub on the client side to invoke a method of a remote object. The stub marshals the method arguments and passes it to the remote reference layer. The remote reference layer converts the request into a single network level request and sends it over the network to the remote object. The remote reference layer receives the request on the server side, which then passes the request to the skeleton. The skeleton unmarshals the arguments and invokes the desired method of the server object. If the method returns a result or an exception, the skeleton object marshals the return value and sends it back to the client object.

Disadvantages:

1) JRMI is based on Java, which is an interpreted language. Hence, it requires more processing power and results in higher cost [Bacellar98].

2) The response times are not bounded. JRMI uses communication protocol such as TCP/IP, which causes non-deterministic behavior.

3) JRMI does not support distribution transparency. The client and server classes have to extend RMI classes and handle exceptions to enable remote object invocation.

2.3.2.3 Distributed Component Object Model (DCOM)

DCOM is a distributed object model from Microsoft Corporation [Williams03]. It is an extension of Component Object Model in order to support remote procedure calls. It is made up of three layers—basic programming layer to provide an illusion to the client that it is invoking methods on objects on the same computer, the remoting layer which contains the COM infrastructure to provide that illusion and the wire protocol which deals with transfer of object requests across the network [Kipfer99].

In order to create an object instance, the client passes its client ID (CLID) to the service control manager on the client side. The service control manager obtains the address of the remote object using the Windows registry and sends the request to create an object instance to the service control manager on the server side. The server-side service control manager loads the object server, requests the object server to create an object reference, creates a stub and passes the reference of the object server instance to the stub. It then passes reference of the stub to the client-side service control manager, which then creates a proxy, passes the stub reference to the proxy and passes the reference of the proxy to the client. The explanation presented here is a simplified version of the actual description.

Once an object instance has been created, the client can then invoke the method of the object instance. The client passes the request to the proxy. The proxy marshals the parameters and sends the method invocation request to the stub. The stub unmarshals the parameters and then calls the desired method [Bacellar98].

Disadvantages:

1) Relies heavily on windows components such as windows registry, file system and installer. All these components placed a heavy demand on already constrained resources of embedded systems. Also, these components have variable execution time.

2) Does not provide distribution transparency. Both client and server side code need to include DCOM APIs to achieve remote procedure invocation [Bacellar98].

In order to summarize, we have listed here the disadvantages of RPC as a means of communication in power electronics control systems.

1) Power Electronic Control system applications require control information to be exchanged asynchronously between controllers. Remote Procedure calls, on the other hand are more suited to synchronous procedure calls. Although CORBA supports asynchronous messaging, it is does so using the architecture designed for synchronous communication. Hence its efficiency is less than what can be achieved with a protocol designed specifically for asynchronous communication.

2) Remote Procedure Calls do not offer distribution transparency. Thus, applications designed for single processor systems cannot be used for multi-processor systems. In [Kipfer99], a new distribution interface has been added to hide the CORBA distribution infrastructure from the application, thereby achieving distribution transparency. However, this transparency is achieved at the expense of efficiency because of the overhead associated with the new layer added between CORBA and the application.

2.3.3 Other Approaches

2.3.3.1 Data Parallel Applications

Roxana Diaconescu in [Diaconescu02-1, Diaconescu02-2, and Diaconescu02-3] and Reidar Conradi in [Diaconescu02-1] have proposed a model to achieve location transparency in loosely synchronous data parallel applications. Large computations in such applications occur independently and consistency is required only at synchronization points.

[Diaconescu02-1] presents an object model to exploit coarse-grain parallelism. The data in this model is divided into sequential and distributed objects. Sequential data objects model trivial data parallel computations. Hence, they do not require consistency and are replicated in the address space of each processor. Distributed data objects, on the other hand, models large application data involved in data intensive operations. They are indicated as distributed by the user. The system partitions the distributed data equally across multiple address spaces using a multi-objective graph partitioning algorithm. Distributed objects are accessed through read and write operations, which are provided by the user.

Data around distributed objects are classified as truly owned data and replicated data. Truly owned data is the data from the data partition assigned to a given processor. Replicated data is the data on a remote processor but needed for computation by the local processor. Copies of replicated data are stored locally on the processors and updated periodically by the system so that processors always get latest values. The system performs loose synchronization for read/write dependencies for distributed objects. The read and write operations are overloaded to ensure that the user have access to truly owned data only.

2.3.3.2 Distributed Oz

[Haridi99] uses logic variables to achieve location and distribution transparency in Mozart Programming System, which implements the Oz language. Oz appears to the programmer as a concurrent object-oriented language with dataflow synchronization. Dataflow behavior is implemented using logic variables.

Conceptually, a logic variable has a fixed value from the moment of its creation. The value of logic variable is not known initially. Once a logic variable is bound, its value becomes available to all the processes using the variable. If a thread needs a value of a logic variable, it gets blocked until a value is assigned to the logic variable. This property can be used to implement dataflow behavior, wherein a thread blocks waiting for data to be made available.

There are 2 basic operations that can be performed on a logic variable – binding a value to a logic variable and waiting until bound. In waiting until bound, the threads waiting on a logic variable get blocked. Once the variable is bound, these threads are awakened. In order to implement binding, each logic variable has an owner site. When a variable is bound to a value, a binding request message is sent to the owner site to inform it of the binding. The owner site then sends messages to all sites that know the variable. The owner site accepts the first binding

request message and ignores subsequent requests. The sites whose binding request messages have been ignored will try again after they receive the binding request.

Logic variables can be used to express four important concurrent programming idioms namely, synchronization, communication, mutual exclusion and first-class channels. In distributed Oz, a FIFO channel can be implemented as a stream, which is a list with an unbound tail. The tail is declared as a logic variable. When a producer writes data in to the channel, it binds the channel to the data item and a new tail. Whenever a new element is added, a message is sent to the consumer informing it of the binding. The consumer can then read from the channel. Usually, a stream is associated with a port in Oz.

2.3.3.3 Generative Communication in Linda

Linda as described in [Gelernter85] and [Schollmeyer91] is a programming language developed in 1980s. It consists of a set of commands that can be added to any programming language to develop the N-Linda programming language.

As per Linda programming model, processes communicate with each other through a shared system buffer known as tuple space. Tuples are sequences of data fields that are added into the tuple space through out() operation and retrieved from the tuple space using in() and rd() operations. The processes are not aware of the locations of other processes, with which they are communicating. Thus Linda does offer transparent communication between distributed processes.

Chapter 3

Protocol Design

Chapter 2 describes the objectives of a transparent inter-processor messaging protocol for dataflow applications and explains why present distributed messaging protocols fail to achieve those objectives. In this chapter, we present a new messaging protocol and explain how it achieves the various design objectives. This chapter also includes a description of different approaches that have been considered in designing the protocol and the reasons for choosing one approach over others.

3.1 Design Overview

As shown in Figure 3.1, the inter-processor messaging protocol is a 6-step protocol.

1. **An ECO writes data to a data channel**: When a source ECO writes data to a data channel with the sink ECO on a different processor, the OS takes the data from the data channel, packs it into packet and writes the packet in a buffer in the FPGA. The OS does not yet delete data from the data channel.

2. **Send packet onto the ring:** The FPGA waits for an empty packet on the ring. When it sees an empty packet, it replaces it with the data packet in the FPGA buffer.

3. **Write data from packet onto data channel:** The FPGA at the destination processor takes the data packet from the ring and interrupts the DSP. The OS unpacks the packet and checks to see if there is space in the data channel identified by the packet. If there is space in the data channel for the entire packet data, the OS writes all the data bytes from the packet onto the data channel. If there is insufficient space, the OS will either overwrite the oldest or the newest data element or write data bytes equal to the available space in the data channel. The actual step taken depends on the value of the `overflow_style` field of the data channel.

4. **Create acknowledgement packet:** The OS then creates an acknowledgement packet, acknowledging the number of bytes written into the data channel. The acknowledgement packet is then written onto the send buffer in the FPGA.

5. **Send packet over the network:** When the FPGA sees an empty packet on the ring, it replaces it with the packet in the FPGA buffer.

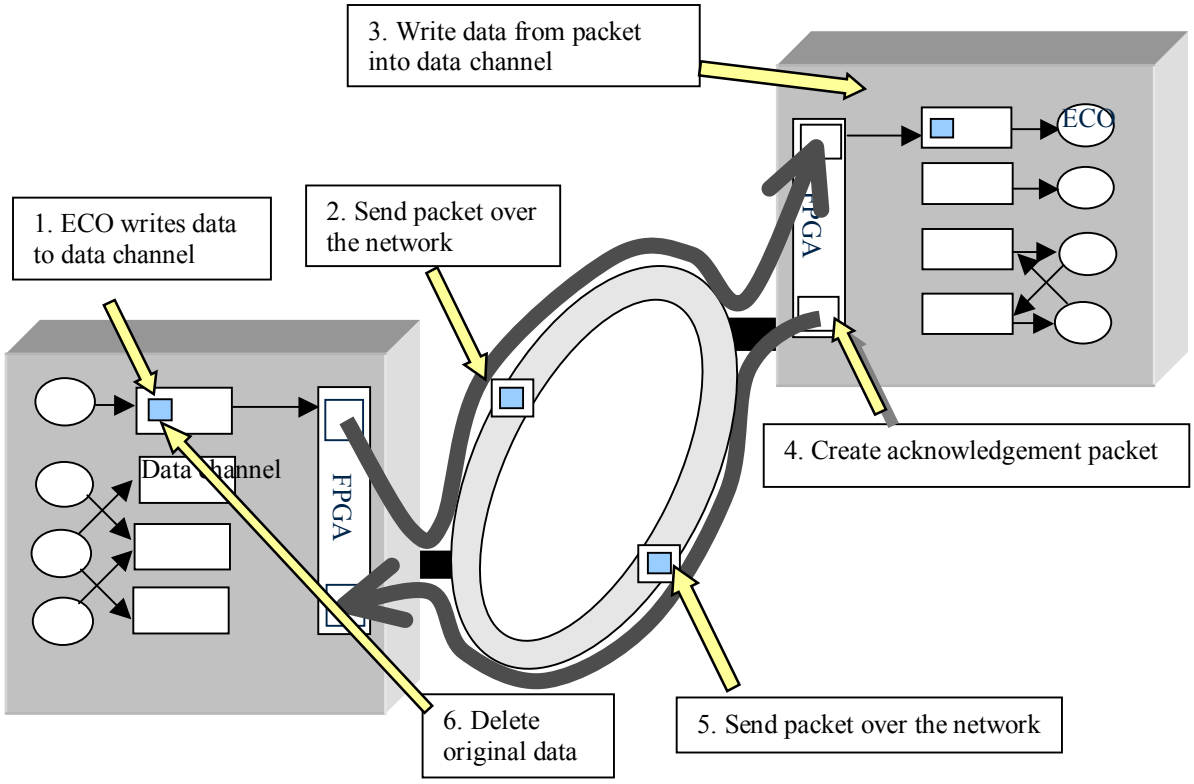


Figure 3.1 Protocol Design

6. **Delete original data:** When the FPGA at the source processor receives the acknowledgement packet, it interrupts the DSP. The OS then removes from the data channel identified by the packet data bytes, equal to the number of bytes acknowledged.

3.2 Protocol Design Elements

The protocol design elements that serve as communication points within the protocol are: distributed data channels, the send queue, the acknowledgement buffer, the FPGA send buffer and the FPGA receive buffer.

Distributed data channel: A distributed data channel has source and sink ECOs on different processors. The structure of a distributed data channel is the same as that of a normal data channel which has source and sink ECOs on the same processor. The nature of a data channel (whether normal or distributed) can be obtained from its `allocation_type` field. This field has a value `NULL` for normal data channels. For distributed data channels, the value can be one of the following - `SENT, WAITING_TO_SEND and EMPTY`.

When the data channel contains data to be sent across the network, the `allocation_type` field has value `WAITING_TO_SEND`. If the data is sent and an acknowledgement is awaited, the `allocation_type` field has value `SENT`. If the data channel neither contains data to be sent across the network nor is waiting for an acknowledgement, its `allocation_type` field has value `EMPTY`.

Send queue: Send queue is a circular queue of pointers to data channel. When an ECO writes to a distributed data channel, a pointer to that channel gets stored in the send queue. Thus, the send queue contains pointers to all the data channels that have data to send across the network. Pointers in the send queue are stored in the decreasing order of the `dist_priority` field value of the data channels. This value is same as the highest priority of the firing rule of the sink process.

Note that at a time, corresponding to a given data channel, there will be only one pointer in the send queue. Thus, the size of the send queue will be same as the number of distributed data channels in the processor.

Acknowledgement buffer: Acknowledgement buffer is a fixed size array of Boolean values with one array index for each distributed data channel. Thus, the size of the acknowledgement buffer is same as the number of distributed data channels. Initially, all the slots in the array have value false. When a data packet from a data channel is sent across the network, the slot corresponding to the data channel is assigned the value true. The value again becomes false when an acknowledgement is received. Note that the send queue and the acknowledgement buffer together will contain not more than one entry corresponding to each distributed data channel.

FPGA send buffer: When the OS processes the send queue, it prepares a data packet in the FPGA send buffer, to be sent to the destination processor. The FPGA send buffer is a fixed-sized buffer to store data and acknowledgement packets to be sent across the network. A packet contains source and destination processor addresses to be used by the FPGA to route the packets to the appropriate destination processor. Rest of the information in the packet is used for reliable transfer of data onto the appropriate data channel at the receiver processor.

The FPGA send buffer contains space for only one packet. Once the OS finishes writing data into the buffer, it enables the transmit flag in the FPGA. This causes FPGA to transfer data into one of its internal buffers to be sent later on the network.

FPGA receive buffer: The FPGA receive buffer is of the same size as the FPGA send buffer. When a packet is received from the network, the FPGA writes it into the FPGA receive buffer, to be read by the OS. If a new packet arrives before the previous one is read by the OS, the new packet will overwrite the previous packet.

3.3 Design Description

The Interprocessor messaging protocol is a 6-step protocol as shown in Figure 3.1. Figure 3.2 describes the steps taken to send a data packet across the network. Figure 3.3 describes steps to receive a data packet and to send back an acknowledgment. The steps taken to delete data from data channel based on an acknowledgement packet is described using figure 3.4.

1) **ECO writes data to data channel:** ECO writes data into the data channel using `write_typed_object()` API. The ECO does not know whether it is writing data to a normal or a distributed data channel. If the ECO writes to an empty distributed data channel, a pointer to the data channel is stored in the send queue, based on the value of its `dist_priority` field. Also, a global variable called `actions_pending` is set to `future_actions` to get the OS to check the send queue.

   The OS checks the send queue to determine if there is data to be sent across the network. In case of a non-empty send queue, the OS makes a copy of the data from the data channel pointed to by an entry in the send queue, packs it into packet and passes the packet onto the FPGA. The pointer to the data channel is then removed from the send queue and an entry in the acknowledgement buffer corresponding to the data channel is set to true. Note that data is not yet removed from the data channel.

2) **Send Packet over the network:** When the FPGA detects an empty slot on the network ring, it replaces it with the packet. The packet contains destination processor address to identify the receiver processor and data channel identifier to identify the data channel on the receiver processor to which the data from the packet is to be written to.

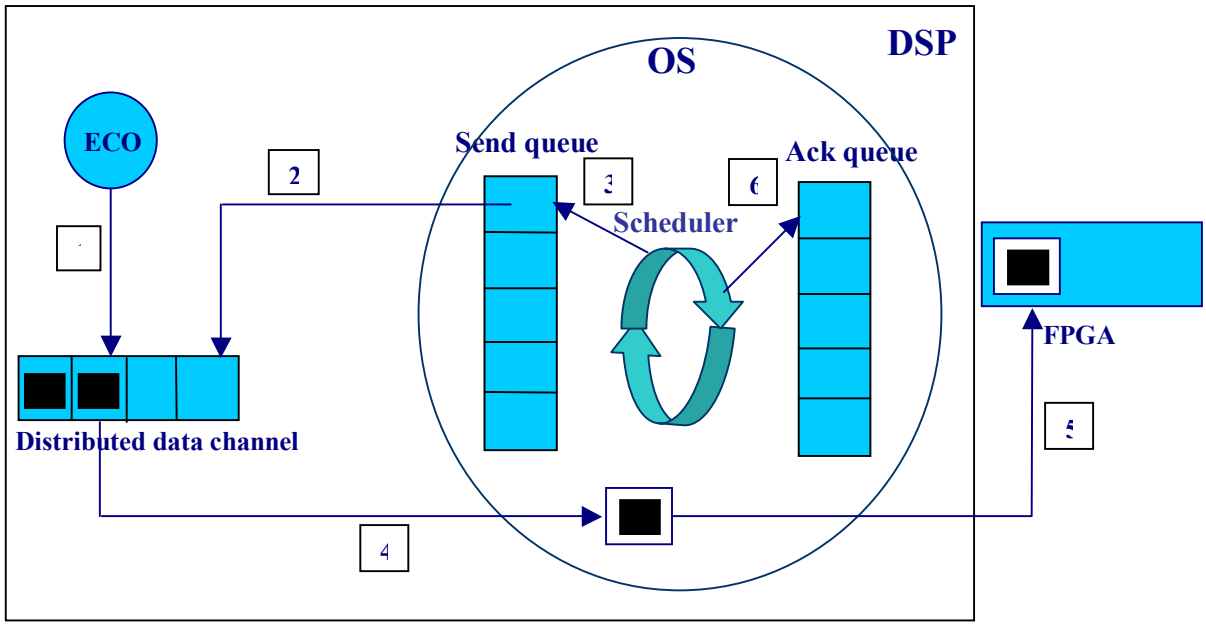
3) **Write data from packet into data channel:** When the FPGA on the receiver processor receives the packet, it interrupts the DSP. The OS unpacks the packet and checks to see if there is space for the data in the data channel, identified by the packet. If there is space for the entire packet, then it writes the data onto the data channel. In case there is insufficient space in the data channel, the OS checks the `overflow_style` field of the data channel. If the `overflow_style` field has value `OS_Overwrite_Oldest` or `OS_Overwrite_Newest`, the OS overwrites the oldest or the newest data element respectively. If the value is `OS_Block`, then the OS writes data bytes that fit into the available space in the data channel.

4) **Create Acknowledgement:** The OS then creates an acknowledgement packet acknowledging the number of bytes written to the data channel. The OS then passes the acknowledgement packet onto the FPGA.

5) **Send packet over the network:** The FPGA waits for an empty slot on the ring before sending the packet.

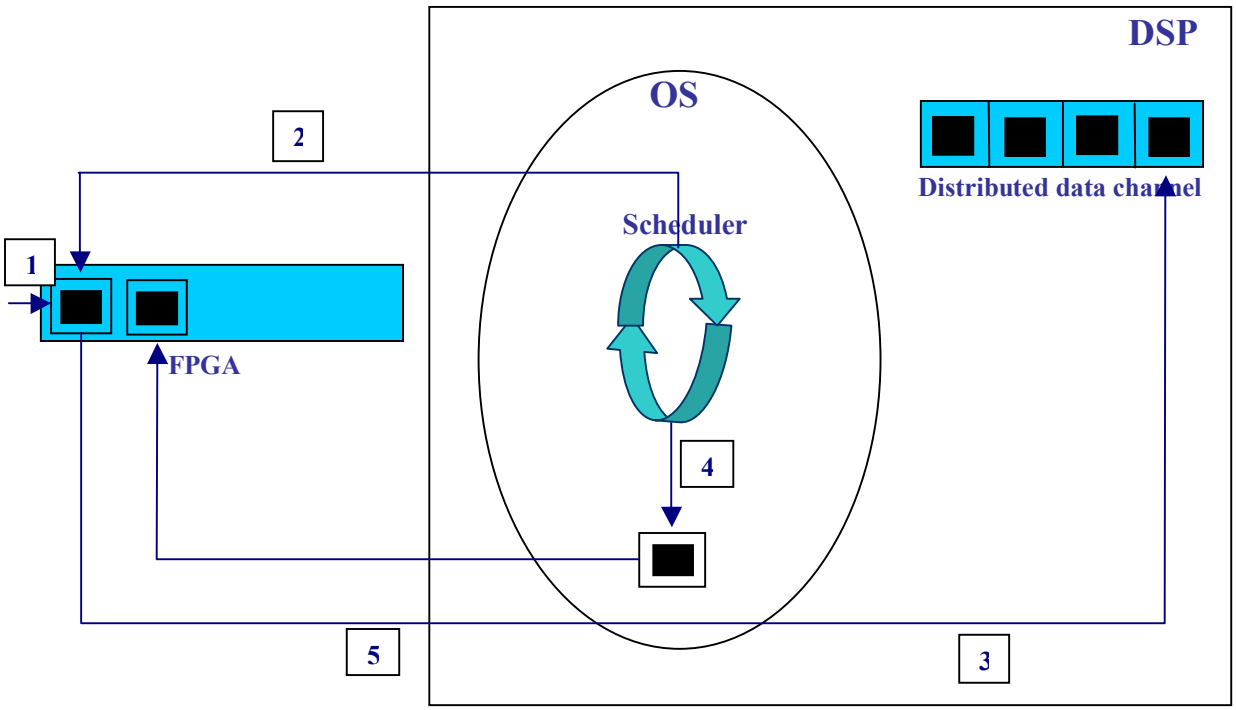
6) **Delete original data:** When the FPGA on the sender side receives the acknowledgement packet, it interrupts the OS. The OS then deletes data from the data channel based on the

number of bytes acknowledged. The entry in the acknowledgement buffer corresponding to the data channel is then set to false.



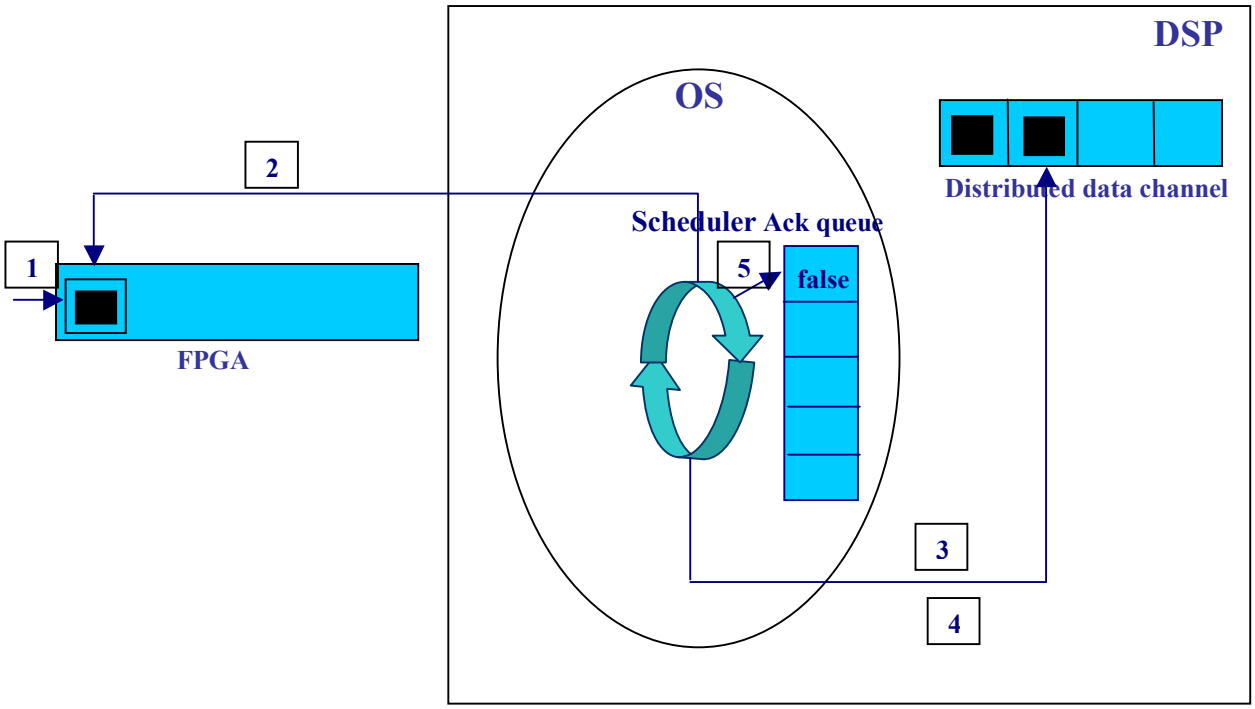
1. ECO writes to a data channel
2. A pointer is added to the send queue
3. OS processes the send queue
4. OS prepares the data packet
5. OS writes data packet into FPGA
6. Sets entry in the ack queue as false

Figure 3.2 Packet send protocol



1. FPGA receives data packet; interrupts the DSP
2. OS processes the data packet, checks for space in the data channel
3. OS writes data from data packet into data channel
4. OS prepares the ack packet
5. OS writes ack packet into the FPGA

Figure 3.3 Packet Receive protocol



1. FPGA receives ack packet; interrupts the DSP
2. OS processes the ack packet
3. OS checks if unacknowledged data has been overwritten
4. OS deletes data based on number of bytes acknowledged
5. OS sets the ack queue entry to false

Figure 3.3 Packet Receive protocol

3.4 Design Decisions

This Section presents the different aspects that were considered while designing the protocol and explains the reasons why we chose one design alternative over another. While considering the various design alternatives, we have given preference to fault tolerance and transparency over efficiency. The power electronics control system needs to be highly reliable. Flexibility is another important factor. To ensure high flexibility, it is necessary that the protocol provides both location and distribution transparency to allow dynamic allocation of processes to processors and easy re-configuration.

The design decisions are classified into the following categories:

- Data Structures used in the protocol
- Transfer of packets from source to destination
- Packet Acknowledgement and
- Insufficient space in the data channels on the sender and receiver side

3.4.1 Data Structures used in the protocol

In this Section, we present the decisions taken regarding the data structures used in the protocol. The protocol makes use of a send queue which contains pointers to data channels that have data to send across the network and acknowledgement queue which indicates data channels that are awaiting acknowledgement.

Send queue implemented as a circular queue
Send queue contains pointers to distributed data channels that have data to be sent across the network. Pointers to data channels in the send queue will be inserted based on the priority of the data channels with the highest priority data channels being inserted at the start of the queue. As the data channels with the highest priority will be processed first, the send queue is implemented as a circular queue.

Acknowledgement buffer as a fixed size array
Acknowledgement buffer is an array of Boolean values. Each entry in the array corresponds to a distributed data channel. If a data channel is waiting for an acknowledgement, its entry in the acknowledgement buffer will have a value true else it will have a value false. An acknowledgement for a packet sent first may arrive after an acknowledgement for the packet sent second. As the order in which the acknowledgements will arrive is not fixed, we have not made use of a queue or a stack like structure.

3.4.2 Transfer of packets from source to destination

There are several factors that need to be considered in order to transfer packets from source to destination. These factors include packet priority, number of messages to be sent per packet and the number of packets to be sent per data channel. Also, as reads and writes from data channels are performed in terms of integral number of data elements, the protocol ensures that the packet always contain data in terms of integral number of data elements. This requires special handling in case of data channels where the size of the packet is less than the size of the data element.

Packet Priority
As scheduling in DARK is done using firing rules, we have set the priorities of the distributed packets based on the highest priority of the firing rule of the sink process that is associated with the distributed data channel. This value is determined at system startup and stored in the dist_priority field of the distributed data channel. For normal data channels, this field has value 0. In case of packets with equal priorities, the packets will be sent out on a first-come-first serve basis.

Transferring only one message in a single packet
We considered packing messages, having the same sender and receiver processors in a single packet given that the packet size is large enough to hold two or more messages. This will definitely help improve performance for applications that have very little information (1-2 bytes) to be exchanged between processes on different processors. In such a case, the source and destination address information will have to be included only once in the packet. However, every packet will have an additional field called num_messages to indicate the number of messages being packed in a single packet.

For simplicity reasons, we are currently transferring messages for only one data channel in a packet. Future work will involve packing multiple messages in a single packet.

One packet per channel at a time
Although multiple packets can be sent from one processor to another at a time, only one packet per data channel can be sent from the sender to the receiver. The next packet for a data channel can be sent only after an acknowledgement is received for the previous packet. This is done to ensure delivery of packets in order. In order to ensure transparent messaging, it is necessary that data be written onto the data channel at the destination processor in the same order as they are written by the source ECO.

Alternate Strategy: Buffering the packets at the sender and the receiver processors and then writing the data in the correct sequence in the data channel can also be done to provide transparent messaging. However, this would require considerable storage buffers and extra processing overhead. If the buffers are maintained by the FPGA, then the data from the packets can be simply copied from the FPGA to the data channel. However, it would require the FPGA to do considerable amount of processing. On the other hand, if the processor maintains the buffers, then the data will have to be copied from FPGA to DSP buffers and then finally from DSP buffers to the data channel which would cause considerable overhead.

Packet size smaller than element size of a data channel
Data stored in a data channel is in the form of integral number of data elements. For e.g. a data channel of type integer will contain an integral number of integers. The `read_typed_object()` and `write_typed_object()` APIs read and write data into the data channel in the form of data elements. Hence, we send data in a packet also as integral number of data elements. This works well for all data channels of primitive data types such as integers, characters and floats, as the packet is assumed to be large enough to store at least one data element. However for data types such as arrays, the packet may not contain space for the entire data element. For such data types, the data elements are split into packets. Thus, a single data element may be spread over multiple packets. We write separate read and write functions for data channels of type arrays to handle data elements split over multiple packets.

When a packet containing part of the data element reaches the destination processor, data from the packet is written onto the data channel, identified by the packet. The `num_entries` field of the data channel indicating the number of data elements in the data channel is computed by dividing the number of data bytes present in the data channel by the size of the data element of the data channel. Hence, until the entire data element is written into the data channel, the value of the `num_entries` field will not be incremented. Instead of calling OS scheduler after the write operation, we call the OS Scheduler when there is change in the value of the `num_entries` field. Since, the `num_entries` field will be incremented only after the entire data element is received, the ECO will be fired only when the entire data element has been written into the data channel. However, an acknowledgement packet will be sent back to the source processor for every data packet received.

When an acknowledgement is received at the source processor, data bytes equal to the number of bytes acknowledged are deleted from the data channel, identified by the acknowledgement packet. The `num_entries` field is computed by dividing the number of bytes in the data channel by the size of the data element. If the number of bytes in the data channel is not an exact multiple of the size of the data element, then the `num_entries` field is assigned the next integer value. For e.g. if the number of bytes in the data channel is 5 and size of data element is 2, then the `num_entries` field is assigned a value of 3. Thus, the `num_entries` field will not be affected until the entire data element has been acknowledged. Since the OS scheduler is called only if there is a change in the `num_entries` field, the scheduler will be called only when the entire data element has been acknowledged.

3.4.3 Packet Acknowledgement

The protocol requires that an acknowledgement be sent for every data packet received and that the data from the data channel at the source processor be deleted only after an acknowledgement is received. This Section discusses the reasons for an acknowledgement for every data packet and deletion of data after the acknowledgment is received.

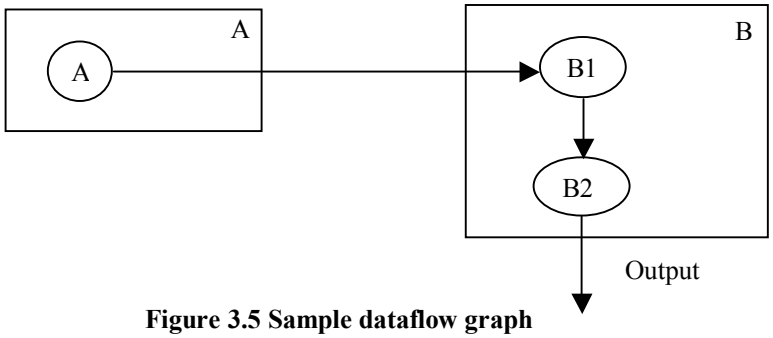


Figure 3.5 Sample dataflow graph

Packet Acknowledgement

In our application, a packet lost due to node or ring failure can seriously affect the working of the system. Consider figure 3.5 where ECO B1 on processor B is waiting for data from ECO A1 in processor A. If the packet from A to B is lost, B1 will fail to fire. As process B2, in order to fire, needs data from process B1 process B2 won't be executed and hence no output will be generated. DRPESNet, the underlying communication protocol, ensures that no packet will be lost due to node or ring failure. However, it does not ensure reliable delivery of data from the packets from the sender to receiver.

The processor may be down or the data channel corresponding to the packet may have insufficient available space and **`OS_Block`** as the overflow policy. Hence, in order to ensure reliable transfer of data between source and sink processes on different processors, we require that an acknowledgement packet be sent for every data packet received. However, the use of an acknowledgement for every data packet greatly affects the efficiency of the system.

Alternate strategy: We considered an alternate strategy to ensure reliable transfer of data between processes on different processors. With this strategy, a buffer with size equal to the size of the data field in a packet is associated with every distributed data channel.

If a packet is sent to a destination processor and the destination processor is down or there is insufficient space in the data channel at the destination processor corresponding to the packet, the packet is sent back to the source processor. The data from the packet is then stored at the source processor in the buffer associated with the data channel corresponding to the packet. An attempt is made to resend the packet at regular intervals. This strategy requires packet associated with a data channel be sent at a specific time interval after the previous packet is sent. If the source processor does not receive the precious packet within the time interval, it is assumed that the packet has been successfully transferred to the sink process at the destination processor. This interval is based on the time it requires for the packet to do a complete round trip. If a processor is down, the network operates in failure mode. During failed mode, the number of hops that a packet must take in order to completely travel around the ring is 2(N-1) incase of link failure and 2(N-2) in case of node failure, where N is the number of nodes in the network. Hence, the worst-case cycle time for a packet is given as

$$T_{cycle} = \left(\frac{s}{1-s} + 2(N-1) + 1 \right) \times \left(\frac{P}{R} + T_{delay} \right)$$

where

T_{cycle} = time to complete one protocol cycle

s = network saturation with value between 0 and 1

N = number of network nodes

P = packet size in bits

R = network speed

T_{delay} = Delay introduced by each network node

The above equation is obtained using the derivation in chapter 5 and replacing the number of nodes N by 2(N-1). This delay may exceed the time required to complete a single switching cycle. For e.g. if number of network nodes N is 8, packet size is 128 bits, network speed is 50Mbps and the delay introduced by each network node is 3ns and assuming network saturation of 90% for worst case delay, the total cycle time can be computed as 40.7517. For an application with a switching period of 50 microseconds, this delay is not acceptable.

Deletion of data after acknowledgement
When a data packet is sent across a network, data is not removed from the data channel, but is stored in the data channel and deleted only after an acknowledgement is received. This is done to provide fault tolerance for packets not received correctly at the receiver processor. Thus, even if a packet is lost, the data in that packet is not lost and that data can be re-sent at some later time.

3.4.4 Insufficient space in the data channels on the sender and receiver side

There may not be sufficient space in the data channel on the receiver side to receive the entire data packet. Also, as data is deleted from the source data channel only after the acknowledgement for that data is received, writes to the source data channel may overwrite unacknowledged data. This Section presents how the protocol handles such situations.

No space on the receiver side
One of the design issues was on how to handle lack of space in the data channel on the receiver processor for the incoming data packet from the sender processor. In case of data channels with overflow policy other than `OS_Block`, the solution is pretty straightforward. We simply overwrite the oldest or the newest data as per the overflow policy of the data channel, as would have been done for intra-processor communication. However, for data channels with overflow policy to block the source process until there is space in the data channel, we considered different alternatives.

Alternative Strategies: One option was to block the source process on the sender processor. However, it was not found to be feasible, as it would require blocking the source process for a long time including the time it takes to send the packet from the sender to the receiver and then for the sink process to read from the data channel and make space for the incoming data. Even if the data channel on the destination processor has space for the incoming data, it would still require blocking the source process until a check is made as to whether there is space in the data channel on the receiver processor.

Another strategy was for the OS at the destination processor to buffer the data. The OS at the destination processor can then write into the data channel when space is available. The OS can then either send the acknowledgement at the time of buffering or at the time of writing data into the data channel. If the OS sends an acknowledgment at the time of writing data into the data channel, the source process may have to wait for a long time before an acknowledgement arrives. The source process may not know if the delay is due to the receiver processor being down or the data channel being full. On the other hand, if the OS sends the acknowledgement at the time of buffering, then the source process will try and send more data packets. If the sink process at the destination processor does not get fired at all due to some reason, the OS may run out of buffer holding the packets.

In our design, we have opted to send back an acknowledgement packet acknowledging the number of data bytes written into the data channel. For e.g. If there is space in the data channel for only 2 data bytes from the packet, we write those 2 bytes into the data channel and send an acknowledgement packet acknowledging 2 bytes. If there is no space in the data channel, we send an acknowledgement packet acknowledging zero bytes. The source process can then attempt to re-send the unacknowledged data. In this case, the source process will block only if there is no space on the data channel both on the sender and the receiver side. Although, this in effect, increases the size of the data channel to twice its specified value, it doesn't really affect the working of the system to a great extent.

Writes on the sender side

When a data packet corresponding to a data channel is sent across the network, a copy of that data is maintained on the data channel, and deleted only after acknowledgement for the data is received. While the acknowledgement is awaited, the source process may attempt to write data into the data channel. If the data channel is full and the overflow policy of the data channel is `OS_Block`, the source process will block waiting to write data in to the data channel. The problem arises when the overflow style is either to overwrite oldest data or to overwrite the newest data.

If the data is overwritten and if data packet is lost, the system behavior is similar to an intra-processor communication where the data is overwritten before being read by the sink ECO. On the other hand, if data gets overwritten and the acknowledgement for that data is received, deleting data from the data channel based on the number of bytes acknowledged can delete new data. The solution to this problem can be divided into 2 categories based on the overflow style of the data channel.

1) Overwrite newest data: For data channels with overwrite style as overwrite newest, the data element at the rear of the data channel may get overwritten again and again by successive writes. Thus, we need to keep track if the last element of the data channel has been overwritten at least once, since the packet has been sent. If the last element has been overwritten at least once, then all the data bytes acknowledged by the acknowledgement packet will be deleted from the data channel except the bytes corresponding to the last element.

2) Overwrite oldest data: In this case, all we have to do is to count number of bytes that have been overwritten, since the packet has been sent. Then when the acknowledgement is received, and the number of bytes acknowledged is less than the bytes overwritten, then the acknowledgement is ignored. If the number of bytes acknowledged is more than the number of bytes overwritten, data bytes equal to the difference between the bytes acknowledged and the bytes overwritten, is deleted from the data channel.

Alternative Strategies: An alternate solution to this problem was to store a copy of the data sent in a separate buffer. However, this would require storage buffers for all the distributed data channels. Also, it would cause considerable overhead copying data for every packet sent.

Another solution was to not allow the ECOs to overwrite data in a full data channel that is awaiting an acknowledgement. However, this would have affected the transparency of the system.

Chapter 4

Protocol Implementation

In this chapter, we present the implementation of the protocol and the manner in which this protocol interacts with the DARK kernel to provide transparent distributed messaging. The protocol is implemented in C. As the protocol is implemented for embedded power electronics control systems, it currently runs on the Analog Devices SHARK 21xxx 32-bit digital signal processors.

In Section 4.1, we present a description of the ECOs and the read and write macros through which they communicate with each other. In Section 4.2, we present the read_typed_object() and write_typed_object() [Singh02-2] API calls and the changes made to them to enable their use for distributed messaging. Section 4.3 presents the APIs used to transfer data from the source ECO on one processor to a destination ECO on another processor across the network.

4.1 Elementary Control Objects

Elementary Control objects or the ECOs [Guo02] are the software building blocks for constructing control applications. A dataflow application can be built by wiring together dataflow components taken from ECO library. An ECO library consists of a .c file and a couple of header files for each ECO. The .c file for an ECO describes the ECO body. The header files for an ECO contains configuration information, firing rules, firing mask, number, and type of input and output ports.

The ECOs are data driven entities. They wait for incoming data items, perform the computation and generate results onto outgoing data channels. Figure 4.1 [Singh02-2] describes a typical pseudo-code implementation for an ECO. The wait_to_fire function checks to see if the data required by the ECO is available. If the data is not available, the ECO is suspended until the necessary data arrives. If the data is available, then the ECO checks to see which of the firing conditions have been satisfied and accordingly read data values, performs computation and then writes results onto the outgoing data channels. Thus, there are no explicit calls to other ECOs. The communication is mainly through data channels. Hence, the ECOs are not aware of the identities and location of other ECOs with which they are communicating.

```
void Sample_ECO_Body (Process_Data* p)
{
    while (wait_to_fire (p))
    {
          switch (p->wakeup_call)
          {
                case SAMPLE_ECO_FIRING_MASK_1:
                      read_type_DC (p->inport[X1], &x1);
                      /* ..action1.. */
                      write_type_DC (p->output[Y1], y1);
                      break;
                case SAMPLE_ECO_FIRING_MASK_2:
                      read_type_DC (p->inport[X2], &x2);
                      /* ..action2.. */
                      write_type_DC (p->output[Y2], y2);
                      break;
                default: break;
          }
    }
}
```

Figure 4.1 Pseudo-code implementation of a typical ECO

```
typedef struct
{
     int               index;
     int               channel_id;
     alloc_type        allocation_type;
     int               difference;
     int               dist_priority;
     Type_Tag          type;
     short int         element_size;
     Array_Descriptor  array_dimensions;
     Overflow_Style    overflow_style;
     volatile int      front;
     volatile int      rear;
     int               size_in_bytes;
     int               size_in_elts;
     volatile int      num_entries;
     volatile bool     blocked;
     bool              interrupt_driven;
     Process*          source_process;
     Process*          sink_process;
     char              buffer [1];
} DC_Queue;
```

Figure 4.2 Data channel representation

The ECOs communicate through unidirectional data channels. The data channels in DARK are typed data channels with support for scalar data types such as integers, floats, double as well as complex data types such as single dimensional byte arrays and multi-dimensional arrays of integers, floats and double data types. The data channels are implemented as circular queues.

The structure of a data channel is as shown in Figure 4.2. The `allocation_type`, `channel_id`, difference and `dist_priority` fields are required to support inter-processor communication. The `allocation_type` field is used to indicate the nature of a data channel (whether normal or distributed). It has a value `NULL` for normal data channels. For distributed data channels, the value can be one of the following—`SENT` for data channels that have already sent data and are awaiting acknowledgement, `WAITING_TO_SEND` for data channels that have data to send across the network and `EMPTY` for distributed data channels that have neither data to send across the network nor are awaiting an acknowledgement. The `channel_id` field serves to uniquely identify the data channel. The `difference` field is used for distributed data channels to indicate the number of data bytes that have been overwritten while an acknowledgement for those data bytes is awaited. The `dist_priority` field gives the priority of the distributed data channels. This priority value decides the order in which packets from different data channels are processed and sent on the network.

The `index` field of the data channel is used for internal operations. The `type` field indicates the type of data that can be stored in the data channel. The `overflow_style` field indicates the action to be taken for a write operation to an already full data channel. Based on the value of the `overflow_style` field, the writing ECO may wait, overwrite the newest data element or overwrite the oldest data element. The `front` and `rear` fields are used for queue management. The `size_in_bytes` field indicate the size of the data channel in bytes. As the data in the data channel is always read or written in terms of data elements and the size of the data channel is always an integral multiple of the data elements, we have a field called `size_in_elts` to indicate the size of the data channel in terms of the number of data elements in the data channel. The `num_entries` field indicates the number of data elements present in the data channel. The `blocked` field has value true if the source or the sink ECO is blocked trying to read data from or write data to the data channel. The `interrupt_driven` field has value true if the source of the data channel is an event handler. It has value false if the source is an ECO. The `source_process` and `sink_process` fields contain pointers to source and sink ECOs respectively.

The number and type of ECOs to be used in an application and the number and type of data channels connecting them is defined using the dataflow descriptor file. The dataflow descriptor file is provided by the user and is used to initialize the system at startup. It contains `DFG_Node_Set` and `DFG_Edge_Set`, which are arrays describing the nodes and edges of a dataflow application. The nodes represent the ECOs while the edges represent the data channels connecting the ECOs. Figure 4.3 [Singh02-2] describes the structures.

```
typedef struct
{
     ECO               eco;
     ECO_Configuration configuration;
     Firing_Rule       *firing_rules;
     int               firing_rule_name;
     Priority          initial_priority;
     Firing_Mask       default_mask;
     unsigned int      assigned_processor;
  /* assigned_processor = processor assigned to the Node          */
} DFG_Node;

typedef struct
{
     Type_Tag         type;
     Node_Number      source;
     Port_Number      source_out_port;
     Node_Number      sink;
     Port_Number      sink_in_port;
     Overflow_Style   overflow_style;
     unsigned int     size;
  /* size = number of elements in the data channel               */
     Array_Descriptor array_dimensions;
     bool             interrupt_driven;
     void*            ISR_signal_value;
     unsigned int     update_frequency;
} DFG_Edge;
```

Figure 4.3 Structures describing the DFG nodes and edges

The `eco` field of the `DFG_Node` provides information such as the number of input and output ports, the weight of the ECO (based on the time it takes to execute), a pointer to the ECO implementation function and the stack size. The `configuration` field specifies the configuration specific information. This information may be different for different applications. The `firing_rules` field is a pointer field pointing to different firing rules through which the ECO can be fired. The `firing_rule_name` field indicates the specific firing rule that will be used to fire the ECO in the current application. The `default_mask` is the firing mask used when the DARK is statically scheduled without any firing rules. The `assigned_processor` field identifies the processor to which this ECO has been assigned.

The `type` field of the `DFG_Edge` structure is used to indicate the type of information written to and read from the data channel. The `source` and `sink` fields indicate the source and sink ECOs while the `source_out_port` and `sink_in_port` identifies the ports on the source and sink ECOs to which the DGF edge is connected. The `overflow_style` indicates whether to overwrite the oldest element, or overwrite the newest element or to block the source ECO if there is an attempt to write data to a full data channel. The `size` field indicates the size of the data channel in terms of the data elements. The `array_dimensions` field specifies the array dimensions for a data channel containing data of type array. A data channel can be of two types—source-driven and interrupt-driven. If a data channel is connected to a source ECO, then it is source-driven else

it is an interrupt-driven channel, written to by the event handlers. The `ISR_signal_value` is a pointer to data for interrupt-driven data channels.

4.2 Interprocessor Messaging

As mentioned in earlier chapters, the inter-processor messaging protocol is a 6-step protocol. This Section provides a detailed description of each of the six steps.

4.2.1 ECO writes data to a data channel

Consider a source ECO attempting to write data to a distributed data channel. The structure of a distributed data channel is same as that of normal data channel and is as shown in Figure 4.2.

Based on the data type of the data channel to which the source ECO is writing data, the appropriate write API is called. For e.g. If the ECO is writing to a data channel of type float, it will call the `write_float_DC()` API. Internally, all the `write_*_DC()` functions are implemented using the `write_typed_object()` macro. The data type information is passed to the macro to indicate the type of data being written. The `write_typed_object()` API is implemented as a macro to reduce the overhead due to function call.

A special type of data channel is the mailbox, which is a data channel of one-element size. Figure 4.4 shows the implementation of the `write_typed_object()` macro for mailboxes, while Figure 4.5 is the implementation for data channels other than mailboxes.

The `write_typed_object()` macro will write into the data channel and update the `num_entries` field of the data channel. The `BLOCK_ON_WRITE` and `BLOCK_ON_WRITE_MAILBOX` checks to see if there is space in the data channel for more values. If the data channel is full, the `BLOCK_ON_WRITE` macro may delete the oldest or the newest data element and return successfully or block the source ECO. The action taken depends on the value of the `overflow_style` field of the data channel. In case of mailboxes, the `BLOCK_ON_WRITE_MAILBOX` macro may either block for `overflow_style` field value of `OS_Block` or return successfully.

The `set_mask_on_write` macro sets the `in_ports_ready` field of the sink process for dynamically scheduled version of DARK. If any of the firing rules of the sink process is triggered, it is inserted into the ready queue. The `WRITE_GOTO_OS` macro for the preemptive

```
#if (USE_MAILBOX)
      #define write_typed_object (dc, type, data)
      {
            QueuePointer qptr = (QueuePointer) dc;
            BLOCK_ON_WRITE_MAILBOX  (qptr);
            *(type *) (qptr->buffer) = data;
            qptr->num_entries = 1;                                  \
            if (qptr->allocation_type != null)                      \
            {                                                       \
                  if (qptr->allocation_type == empty)               \
                  {                                                 \
                        insert_send_queue(qptr);                    \
                        qptr->allocation_type = waiting_to_send;    \
                        mod_plus_1(sendq_rear, NUM_DFG_EDGES);      \
                        actions_pending = future_actions;           \
                  }                                                 \
                  else if (qptr->allocation_type == sent)           \
                        qptr->difference = qptr->element_size;      \
            }                                                       \
            setmask_on_write(qptr);
            WRITE_GOTO_OS (qptr);
}
```

Figure 4.4 write_typed_object implementation for all mailboxes

version of DARK checks if the process at the head of the ready queue has the same or higher priority than the current process.

The shaded region indicates the code added to the `write_typed_object` macro to support inter-processor communication. For writes to a distributed data channel that is empty, a pointer to the data channel is added to the send queue using the `insert_send_queue` macro. The `actions_pending` variable is then set to `future_actions`. The `future_actions` value of the `actions_pending` variable will cause the OS to check the send queue.

```
#define write_typed_object(dc, type, data)                              \
 {                                                                      \
      QueuePointer qptr = (QueuePointer) dc;                            \
      int local_front;                                                  \
      int local_rear;                                                   \
      int src_processor;                                                \
      int sink_processor;                                               \
      int OW_Oldest=0;                                                  \
      /*flag set to 1 if overflow style is OW_Oldest */                 \
      BLOCK_ON_WRITE (qptr, type, OW_Oldest)                            \
      local_front = qptr->front;                                        \
      local_rear = qptr->rear;                                          \
                                                                        \
      *(type*) (qptr->buffer + local_rear) = data;                      \
            local_rear = (local_rear + sizeof (type));                  \
      if (local_rear == qptr->size_in_bytes)                            \
      {                                                                 \
            local_rear = 0;                                             \
      }                                                                 \
      qptr->num_entries++;                                              \
      if (qptr->allocation_type! = null)                                \
      {                                                                 \
            if (qptr->allocation_type == empty)                         \
            {                                                           \
                  insert_send_queue (qptr);                             \
                  qptr->allocation_type = waiting_to_send;              \
                  mod_plus_1(sendq_rear, NUM_DFG_EDGES);                \
                  actions_pending = future_actions;                     \
            }                                                           \
            else if (qptr->allocation_type == sent)                     \
            {                                                           \
                  if (OW_Oldest == 1)                                   \
                        (qptr->difference)+=sizeof(type);               \
                  else                                                  \
                        qptr->difference = sizeof(type);                \
            }                                                           \
      }                                                                 \
      qptr->front = local_front;                                        \
      qptr->rear = local_rear;                                          \
      setmask_on_write(qptr);                                           \
      WRITE_GOTO_OS (qptr)                                              \
  } #endif
```

Figure 4.5 Implementation of a write_typed_object macro

In case of writes to a distributed data channel that is awaiting an acknowledgement, the `difference` field of the data channel is updated to reflect the number of data bytes that have been overwritten. If the overflow style of the data channel is to overwrite the newest data element, then only the last element will get overwritten, irrespective of the number of writes performed. Hence, for overflow style as overwrite newest, the difference field is assigned the size of one data element. If the overflow style is to overwrite the oldest data element, then we need to keep track of the number of data elements that have been overwritten since the packet is sent. For mailbox data channel, the number of bytes that can be overwritten will not exceed the size of one data element. Hence, the difference field in case of overflow for mailbox data channel is always set to the size of one data element.

Figure 4.6 shows the `send_queue` structure. The size of the `sendq` is determined at run time based on the number of distributed data channels. The `insert_send_queue` macro inserts pointers to data channels in decreasing order of their distribution priority. Its implementation is as shown in Figure 4.7.

When the OS dispatcher is called, it will check the value of the `actions_pending` variable. If this value is set to `future_actions`, it will check the `send_queue` to see if any data channels have data to be sent across the network.

If there are entries in the `send_queue`, the `pesnet_send_handle` macro is called. The macro checks for space in the FPGA send buffer, which is determined based on the value of a bit in a control register. The bit has value one when the send buffer is non-empty and has packet to be sent across the network, else the control bit has value zero. If the send buffer is empty, the `read_bytes_packet` macro is called to pack data from the data channel into a packet and to write the packet into the FGPA send buffer. A packet consists of two parts—command and data.

```
typedef struct
{
      DC_Queue **sendq;
 } Send_Queue;
```

Figure 4.6 Send queue structure

```
void insert_send_queue (QueuePointer qptr)
{
    int i;
    i=sendq_rear;
    if (i == sendq_front)
      send_queue.sendq[sendq_front] = qptr;
    else if (i > sendq_front)
    {
      while (qptr->dist_priority >
            send_queue.sendq[i-1]->dist_priority)
      {
            send_queue.sendq[i] = send_queue.sendq[i-1];
            i--;
            if (i == sendq_front)
                  break;
      }
      send_queue.sendq[i] = qptr;
    }
    else
    {
      while (qptr->dist_priority >
            send_queue.sendq[i-1]->dist_priority)
      {
            send_queue.sendq[i] = send_queue.sendq[i-1];
            i--;
            if (i==0)
            {
                  i = NUM_DISTR_EDGES;
                  if (qptr->dist_priority <
                        send_queue.sendq[i-1]->dist_priority)
                  {
                        i=0;
                        break;
                  }
            }
      }
      send_queue.sendq[i] = qptr;
    }
}
```

Figure 4.7 Implementation of insert_send_queue macro

```
typedef struct
 {
      int* command: 4;
      int* from_address: 8;
      int* to_address:   8;
 } Command_Packet;

 typedef struct
 {
      int*        number_of_bytes: 4;
      int*        channel_id:  8;
      int*        data[PACKET_SIZE];
 } Data_Packet;
```

Figure 4.8 Packet Structure

```
typedef struct
{
      unsigned int* cmd_data;
      /* cmd_data has fields
            int* command:           4;
            int* number_of_bytes:   4;
            int* channel_id:        8;
            int* from_address:      8;
            int* to_address:        8;
      */
      int data[PACKET_SIZE];
 } Combined_Packet;
```

Figure 4.9 Implemented Packet Structure

The command and data information in a packet is specified as per structures `Command_Packet` and `Data_Packet` as shown in Figure 4.8. The `command` field indicates the packet type. The `from_address` and `to_address` fields give the source and destination processor addresses respectively. The `number_of_bytes` field indicates the number of data bytes stored in a packet for a data packet. For an acknowledgement packet, the field specifies the number of data bytes being acknowledged. The `channel_id` field identifies the distributed data channel for which the packet is being sent. The `data` field stores the data for a data packet. In case of an acknowledgement packet, this field is ignored.

While implementing the protocol, we have clubbed data and command fields of a packet into a single structure as shown in Figure 4.9. This is done so that packets can be written to the FPGA Send buffer using fewer write statements.

```
extern Combined_Packet send_packet;
#define read_bytes_packet(qptr)                                          \
{                                                                        \
      int i;                                                             \
                                                                         \
      log ("read typed object: transferring data\n");                    \
      for (i=0; i<qptr->element_size; i++)                               \
          *(send_packet.data[i]) = *(qptr->buffer+qptr->front + i);\
                                                                         \
      *(send_packet.cmd_data) = data_packet | proc1_shift |              \
        (qptr->element_size << 24) | (qptr->channel_id << 16) |          \
        DFG_Node_Set [DFG_Edge_Set                                       \
        [qptr->channel_id].sink].assigned_processor;                     \
                                                                         \
      *transmit_enable = 1; /*transmit enable is a flag to indicate\
      to the FPGA that data has been written into the FPGA send          \
      buffer*/                                                           \
      qptr->allocation_type=sent;                                        \
      ack_queue [qptr->channel_id] = true;                               \
  }
```

Figure 4.10 Implementation of read_bytes_packet for mailbox

Figure 4.10 describes the implementation of `read_bytes_packet` macro for a mailbox data channel. For data channels other than mailboxes, we have considered two cases:

- data channels in which the number of data bytes is 1 and
- data channels in which the number of bytes is more than 1.

Figure 4.11 describes the implementation for data channels other than mailboxes, with number of data bytes as 1 while Figure 4.12 describes the implementation for data channels with number of bytes as more than 1. The variable `send_packet` has fields `cmd_data` and `data[]`, which are pointers to the `cmd_data` and `data[]` fields in the FPGA send buffer. The `read_bytes_packet` macro writes command and data information into the FPGA send buffer. Once a data packet is

```
unsigned int rem_capacity;                                          \
int size;                                                           \
int pack_bytes;                                                     \
                                                                    \
size = ((qptr->num_entries)*(qptr->element_size));                  \
                                                                    \
if (size >= PACKET_SIZE)                                            \
     pack_bytes = (PACKET_SIZE/qptr->element_size)*                 \
          qptr->element_size);                                      \
else                                                                \
     pack_bytes = size;                                             \
                                                                    \
*(send_packet.cmd_data) = data_packet | proc1_shift |               \
     (pack_bytes << 24) | (qptr->channel_id << 16) |                \
     DFG_Node_Set [DFG_Edge_Set                                     \
     [qptr->channel_id].sink].assigned_processor;                   \
                                                                    \
rem_capacity = qptr->size_in_bytes - local_front;                   \
if (rem_capacity >= pack_bytes)                                     \
{                                                                   \
     int i;                                                         \
     for (i=0; i<pack_bytes; i++)                                   \
        packet_data->data[i] =*(qptr->buffer + qptr->front + i);    \
}                                                                   \
else                                                                \
{                                                                   \
     int i;                                                         \
     for (i=0; i< rem_capacity; i++)                                \
        packet_data->data[i] = *(qptr->buffer + qptr->front + i);\
     for (i=0; i < (pack_bytes-rem_capacity); i++)                  \
        packet_data->data [i + rem_capacity]=*(qptr->buffer + i);\
}                                                                   \
*transmit_enable = 1;                                               \
/* transmit enable is a flag to indicate to the FPGA that data     \
   has been written into the FPGA send buffer*/                     \
qptr->allocation_type=sent;                                         \
ack_queue [qptr->channel_id] =true;                                 \
```

Figure 4.11 Implementation of read_bytes_packet for data channels other than mailboxes with number of bytes greater than 1

```
if(((qptr->num_entries) * (qptr->element_size)) == 1)              \
{                                                                  \
      *(send_packet.data[0]) = *(qptr->buffer + qptr->front);      \
      *(send_packet.cmd_data) = data_packet | proc1_shift |        \
            (0x1000000) | (qptr->channel_id << 16) |               \
            DFG_Node_Set [DFG_Edge_Set                             \
            [qptr->channel_id].sink].assigned_processor;           \
 }                                                                 \
 *transmit_enable = 1; /* transmit enable is a flag to indicate to\
  the FPGA that data has been written into the FPGA send buffer*/ \
  qptr->allocation_type=sent;                                      \
  ack_queue[qptr->channel_id]=true;                                \
```

Figure 4.12 Implementation of read_bytes_packet for data channels other than mailboxes with number of bytes as 1

written into the FPGA send buffer, the `allocation_type` field of the data channel is changed to `SENT` and an entry in the `ack_queue` corresponding to the data channel is set to true. The `pesnet_send_handle` macro then sets the `transmit_enable` flag to 1.

Note that although data is read from the data channel, the front pointer of the data channel is not updated. Hence the data is not yet deleted from the data channel. The front pointer is updated when the acknowledgement for the data is received.

4.2.2 Send packet onto the ring

Once the `transmit_enable` flag is set to 1, the FPGA transfers the data from the FPGA send buffer to one of its internal buffer. When there is an empty packet on the ring, the FPGA replaces the empty packet with the packet in its internal buffer.

4.2.3 Write data from packet onto data channel

When the FPGA on the destination processor receives a packet, it interrupts the DSP. In the current version of DARK, an ISR is called for a general-purpose interrupt that logs incoming event into the event queue. After that, the control returns back to the point where it was interrupted. The interrupt is processed later when the dispatcher is called.

When the dispatcher is called, it checks the events in the event queue and processes the interrupt. If the interrupt is due to a data packet, the `pesnet_receive_handle` macro is called. The macro checks if there is space in the FPGA send buffer for an acknowledgement packet. If the send buffer is empty, the macro calls `write_bytes_packet` macro to copy the data from the packet onto a data channel identified by the packet and to prepare an acknowledgement packet to be sent back to the source processor.

```
#define write_bytes_packet (qptr, receive)                          \
{                                                                   \
      log ("Writing to mailbox\n");                                 \
      PACKET_BLOCK_ON_MAILBOX (qptr, receive, packet_data)          \
      *ctrl_reg_1 |= 1;                                             \
}
```

Figure 4.13 Implementation of write_bytes_packet for mailbox data channels

```
extern Command_Packet send_packet
#define write_bytes_packet(qptr, receive)                           \
{                                                                   \
     int avail_bytes;                                               \
                                                                    \
     /* Find the available bytes in the data channel            */\
     avail_bytes = qptr->size_in_bytes –                            \
          (qptr->num_entries * qptr->element_size);                 \
                                                                    \
     if ((avail_bytes==0) && (qptr->overflow_style == OS_Block)) \
     {                                                              \
          /*Prepare acknowledgement packet acknowledging
               0 bytes */
          *ctrl_reg_1 |= 1;                                         \
     }                                                              \
     else                                                           \
     {                                                              \
          int OW_Newest = 0; /* flag to indicate if the overflow\
                               style is overwrite_newest */        \
          packet_bytes = ((*(receive.cmd_data)) & 0x0F000000)       \
                               >> 24;                               \
                                                                    \
          PACKET_BLOCK_ON_WRITE (qptr, packet_bytes,                \
               avail_bytes, OW_Newest)                              \
          /*Prepare acknowledgement packet acknowledging
               data bytes equal to packet_bytes */
          *ctrl_reg_1 |= 1;                                         \
          if (packet_bytes == 1)                                    \
          {                                                         \
               /* Transfer one byte of data */
          }                                                         \
          else                                                      \
          {                                                         \
               if (OW_Newest == 1)                                  \
                    /* Transfer only the last packet element
                         onto the data channel*/
               else
                    /* Transfer all elements from the
                         packet to the data channel*/
          }                                                         \
          setmask_on_write (qptr);      \
     }     \
}
```

Figure 4.14 Implementation of write_bytes_packet for data channels other than mailboxes

Implementation of `write_bytes_packet` macro for mailbox and normal data channels is described using Figure 4.13 and 4.14 respectively.

The macro `PACKET_BLOCK_ON_MAILBOX` for dynamically scheduled application checks for available space in the mailbox and its overflow policy. In case if the mailbox is full and the overflow policy is `OS_Block`, the macro prepares an acknowledgement packet acknowledging zero bytes. In all other cases, the macro writes data into the mailbox and prepares an acknowledgement packet acknowledging bytes equal to the element size of the mailbox.

In case if the application is statically scheduled, the macro simply writes data from the packet into the mailbox and prepares an acknowledgement packet acknowledging bytes equal to the element size of the mailbox without checking the available space or the overflow policy. This is so because for statically scheduled application, it is assumed that there will always be space in the mailbox for the incoming data.

The macro `PACKET_BLOCK_ON_WRITE` checks for availability in the data channel. If there is insufficient space in the data channel for the data packet and the overflow policy is `OS_Overwrite_Newest` or `OS_Overwrite_Oldest`, the macro deletes the newest element or the oldest elements respectively. For overflow policy of `OS_Overwrite_Newest`, the `OW_Newest` variable is set to one. If the overflow policy is `OS_Block`, then the macro updates packet_bytes to set it equal to the available space.

4.2.4 Create acknowledgement packet

Once the acknowledgement packet is prepared, the `write_bytes_packet` macro sets the `transmit_enable` flag to 1 indicating to the FPGA that the packet is ready to be sent onto the network.

4.2.5 Send packet over the network

The FPGA moves the packet from the send buffer to its internal buffer. The FPGA waits for an empty packet on the network and replaces it with the packet in its internal buffer.

4.2.6 Delete original data

When the FPGA at the source processor receives the acknowledgement packet, it interrupts the DSP. The OS calls the `delete_ack_bytes` macro to remove data bytes from the data channel based on the number of bytes acknowledged. Figure 4.15 describes the implementation of `delete_ack_bytes` macro for mailboxes.

```
#define delete_ack_bytes (qptr, num_of_bytes)                          \
{                                                                      \
      ack_queue [qptr->channel_id] = false;                            \
      if (qptr->difference != qptr->element_size)                      \
      {                                                                \
            qptr->num_entries = 0;                                     \
            /* Resetting the mask                                   */\
            (qptr->sink_process)->in_ports_ready &=                    \
                ~ (1<<DFG_Edge_Set [qptr->index]. sink_in_port);       \
            if (qptr->overflow_policy == OS_Block)                     \
                  FREE_SOURCE_PROCESS (qptr);                          \
      }                                                                \
      else                                                             \
      {                                                                \
            qptr->allocation_type=waiting_to_send;                     \
            insert_send_queue (qptr);                                  \
            mod_plus_1 (send_queue.rear, MAX_SEND_QUEUE_LENGTH);       \
            qptr->difference = 0;                                      \
      }                                                                \
 }
```

Figure 4.15 Implementation of delete_ack_bytes for mailboxes

The `difference` field of the data channel indicates if the data being acknowledged has been overwritten. If it is overwritten then the acknowledgement packet is ignored. Otherwise data is deleted from the data channel and `FREE_SOURCE_PROCESS` macro is called to wake up any process sleeping on this data channel. Note that if data is overwritten, a pointer to the mailbox is stored in the send queue and the `allocation_type` of mailbox is changed to `WAITING_TO_SEND`.

For data channels other than mailboxes, the number of data bytes to be deleted from the data channel is determined based on the overflow policy of the data channel and the value of the `difference` field. In case of data channels with the value of the `difference` field as 0 or overflow policy as `OS_Block`, the macro simply deletes the data based on the number of bytes acknowledged as shown in Figure 4.16. Figure 4.17 shows the steps taken for data channels with

```
if ((qptr->overflow_style == OS_Block) || (qptr->difference == 0)  \
{                                                                  \
      /* Delete from the data channel data equal to the bytes
            acknowledged                                         */
      if (qptr->num_entries==0)                                    \
            qptr->allocation_type=empty;                           \
      else                                                         \
      {                                                            \
            qptr->allocation_type=waiting_to_send;                 \
            insert_send_queue (qptr);                              \
            mod_plus_1 (send_queue.rear, MAX_SEND_QUEUE_LENGTH);   \
      }                                                            \
   }
```

Figure 4.16 Deletion of bytes from data channel for overflow policy of OS_Block or difference field as zero

```
 if (qptr->overflow_style == OS_Overwrite_Newest)                    \
{                                                                    \
    if (num_of_bytes <= (qptr->size_in_bytes-qptr->difference))      \
    {                                                                \
          /* Delete from the data channel data equal to
                num_bytes                                          */
    }                                                                \
    else                                                             \
    {                                                                \
          /* Delete from the data channel data equal to
                difference between qptr->size_in_bytes
                and qptr->difference                               */
    }                                                                \
    qptr->difference = 0;                                            \
    qptr->allocation_type=waiting_to_send;                           \
    send_queue.send_queue [send_queue.rear] = qptr;                  \
    mod_plus_1 (send_queue.rear, MAX_SEND_QUEUE_LENGTH);             \
}                                                                    \
else                                                                 \
{                                                                    \
    if (num_of_bytes > qptr->difference)                             \
    {                                                                \
          /* Delete from the data channel data equal to difference
          between num_of_bytes and qptr->difference                */
          qptr->difference = 0;                                      \
          qptr->allocation_type=waiting_to_send;                     \
          send_queue.send_queue [send_queue.rear]=qptr;              \
          mod_plus_1 (send_queue.rear,MAX_SEND_QUEUE_LENGTH);        \
    }                                                                \
    else                                                             \
    {                                                                \
          qptr->difference = 0;                                      \
          ack_queue [qptr->channel_id]=false;                        \
          qptr->allocation_type=waiting_to_send;                     \
          send_queue.send_queue [send_queue.rear] = qptr;            \
          mod_plus_1 (send_queue.rear, MAX_SEND_QUEUE_LENGTH);       \
          return;                                                    \
    }                                                                \
}                                                                    \
ack_queue [qptr->channel_id]=false;                                  \
FREE_SOURCE_PROCESS (qptr);                                          \
```

Figure 4.17 Implementation of delete_ack_bytes for data channels other than mailboxes

a non-zero `difference` field value or with overflow policy as `Overwrite_Newest` or `Overwrite_Oldest`. As the difference field is non-zero, the data channel contains new data to be sent across the network. Hence, a pointer to the data channel is stored in the send queue and the allocation type is changed to `WAITING_TO_SEND`. The entry in the acknowledgement buffer corresponding to the data channel is set to false and `FREE_SOURCE_PROCESS` macro is called to wake up any process waiting to write data onto the data channel.

Chapter 5

Network Analysis

In this chapter, we present an analysis of the network performance; based on factors such as network speed, network saturation and number of network nodes.

5.1 Network Analysis

We present network performance as the time required to execute a complete protocol cycle, i.e., the total time required to send a packet from the sender to the receiver, to process the packet at the receiver, to send back an acknowledgement packet, and for the acknowledgement packet to reach the sender.

Thus,

$$T_{cycle} = T_{send} + T_{process} + T_{ack}$$

where

T_{cycle} = time for a complete protocol cycle

T_{send} = time taken to send the packet from sender to receiver

$T_{process}$ = time taken to process the data packet at the receiver

T_{ack} = time to send the acknowledgement packet from receiver to sender

Consider a network consisting of N nodes with D being the number of hops between the sender and the receiver. Let P be the size of the packet in bits and R be the network speed in bits/sec.

The network can be considered as divided into slots of data packets. A packet can be sent over the network only at the start of a packet slot. For a sender to send a packet on the network, it must wait for an empty packet slot before sending the packet. We define saturation delay as the delay caused by network saturation, which is nothing but the time difference between the time when the sender is ready to send the packet and the start of an empty packet slot.

Thus, the time to send a network packet depends on the saturation delay, the time required to send the packet over a single link and the distance between the sender and receiver nodes.

Thus,

$$T_{send} = T_{sat\_delay} + D \times T_{packet}$$

where,

T_{sat_delay} = delay caused by network saturation

T_{packet} = time taken to send a packet over a single network link

Since packets will always be sent out at the start of a packet slot, we express the time to process a packet at the receiver in terms of the number of packet slots. If C is the number of packet slots that pass through the receiver before it is ready to send the acknowledgement packet, then

$$T_{process} = C \times T_{packet}$$

Since the time to process a packet at the receiver is very small, it can be safely assumed that the processing time is equal to length of a single packet slot and hence equal to the time to transmit a packet over a single link.

The time to send back an acknowledgement is given as

$$T_{ack} = T_{sat\_delay} + (N - D) \times T_{packet}$$

Therefore, the time required for a complete cycle can be given as

$$\begin{aligned} T_{cycle} &= T_{send} + T_{process} + T_{ack} \\ &= 2 \times T_{sat\_delay} + (N - D + D) \times T_{packet} + C \times T_{packet} \\ &= 2 \times T_{sat\_delay} + (N + C) \times T_{packet} \end{aligned}$$

If s is the saturation index of the network with value between 0 and 1, then the saturation delay can be given as

$$T_{sat\_delay} = \frac{1}{2} \times \frac{s}{1 - s} \times T_{packet}$$

The time to transmit a packet over a single network link can be given as

$$T_{packet} = \frac{P}{R} + T_{delay}$$

where,
T_{delay} = Delay introduced by each node in the network

Thus, the total cycle time can be give as

$$\boxed{T_{cycle} = \left(\frac{s}{1 - s} + N + C \right) \times \left(\frac{P}{R} + T_{delay} \right)}$$

To provide a basis for concrete discussion, we consider an example application with two controllers and 6 phase legs. The controllers are switching at a frequency of 20KHz. As there are 2 controllers and 6 phase legs, the number of nodes in the network is N=8. Let us assume that the

network saturation is 25% (s = 0.25). As 8 data bits get transmitted as 10 bits due to 4B/5B encoding by the transceivers, for a packet of size 16 bytes, the value of P is 160 bits. If the network speed R is 100Mbps, and the delay introduce by each node is 3 nanoseconds, then

$$T_{cycle} = T_{send} + T_{process} + T_{ack}$$

Let us assume that the 2 universal controllers are separated by 4 nodes. Thus D has a value of 4. Hence, the time to send a single packet from the source processor to the destination processor is given as:

$$T_{send} = T_{sat\_delay} + D \times T_{packet}$$

$$T_{send} = (\frac{1}{2} \times \frac{s}{1-s} + D) \times (\frac{P}{R} + T_{delay})$$

$$= 6.679 \text{ µsecs}$$

Assuming that the time to process a packet at the receiver is equal to the length of a single packet slot,

$$T_{process} = T_{packet}$$

$$= (\frac{P}{R} + T_{delay})$$

$$= 1.603 \text{ µsecs}$$

The time to send the acknowledgement packet from the receiver back to sender is given as:

$$T_{ack} = T_{sat\_delay} + (N - D) \times T_{packet}$$

$$= (\frac{1}{2} \times \frac{s}{1-s} + (N - D)) \times (\frac{P}{R} + T_{delay})$$

= 6.679 µsecs

Hence,

$$T_{cycle} = 6.679 + 1.603 = 6.679$$

= 14.961 µsecs

Thus, in a single switching cycle of 50 microseconds, it is possible to complete 3 cycles of inter-processor communication.

5.2 Validating the Network Analysis

Section 5.2 describes the actions that need to be performed in order to validate the theoretical analysis, the work done so far, the work that still needs to be carried out and the hardware problems that need to be solved in order to do the complete validation.

5.2.1 Steps to validate the theoretical analysis

Section 5.2.1 presents a detailed description of the steps to be performed in order to validate the theoretical analysis.

Emulate communication between two processes on different processor
Prior to testing the protocol on the actual hardware, we need to test the protocol using the VisualDSP++ 3.0 emulator. In order to ensure proper working of the protocol, we need to first test the protocol for communication between two processes on different processors. As the VisualDSP++ emulator does not support multiple processors, we need to modify the code such that although the two processes are on the same processor, it appears as if they are on different processors and communicating with each other through the FPGAs.

Emulate 3-phase boost rectifier application
Figure 5.1 presents a dataflow graph of a closed-loop control for a 3-phase boost rectifier application [Singh02-2]. The rectifier consist of both the voltage and current loops. At the beginning of each switching period, the current ADC driver ECO and the voltage ADC driver ECOs are fired to read feedback information. The voltage loop is executed first to generate references for the current loop.

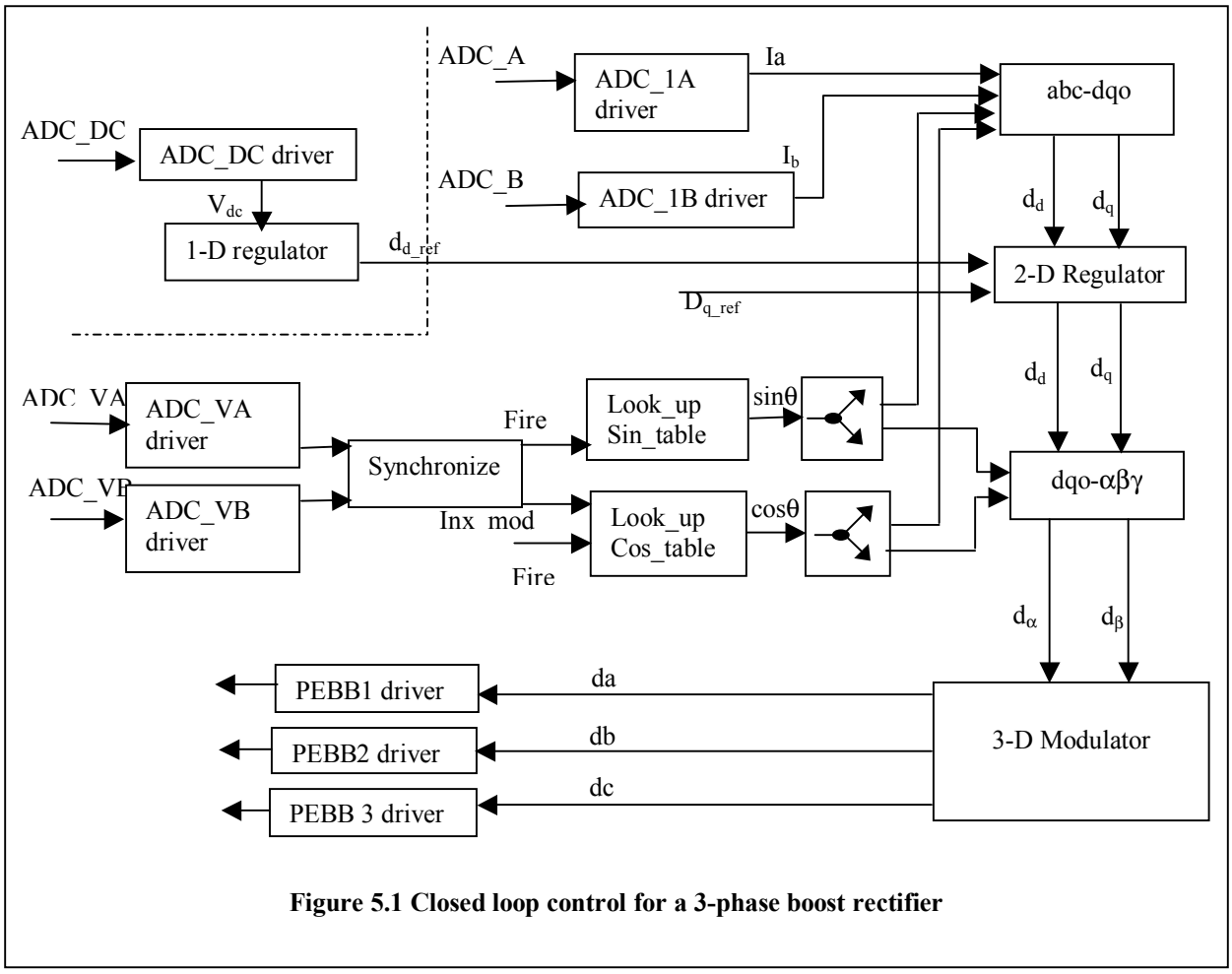


Figure 5.1 Closed loop control for a 3-phase boost rectifier

Emulating the 3-phase boost rectifier on the VisualDSP++ 3.0 emulator is the next step to ensure proper working of the protocol. Again, the code will have to be modified so as to give an impression that the current and the voltage loops are on different processors and communicating through the FPGAs.

Implementing the protocol on two universal controllers for a two-process application

The next step in the protocol validation is implementing the protocol on actual hardware and then executing a control application consisting of two processes on two separate processors. The time required to complete one cycle of the protocol will then be measured and compared with the theoretically obtained value.

Implementing the protocol on two universal controllers for a boost rectifier application

We next plan to implement the protocol for a boost rectifier application. This will involve connecting two processors and three phase legs by a dual fiber optic ring and then measuring the time required for a complete cycle of the protocol for communication between the current and voltage ECOs on different processors. This time will then be compared with the theoretically obtained value.

5.2.2 Amount of work done

As the hardware was not ready at the time of writing this thesis, we have not been able to perform all the steps as mentioned in Section 5.2.1. In this Section, we present the work done so far in order to validate the protocol.

Emulate communication between two processes on the same processor
Using the VisualDSP++ 3.0 emulator, we tested the protocol for communication between two processes. As the two processes were executed on the same processor, we modified the dataflow descriptor file to give an impression that the two processes were on different processors. We also added a small transfer function to transfer data from the FPGA send buffer to the FPGA receive buffer.

Based on the dataflow descriptor file, the data channel connecting the two processes was identified to be of distributed nature. Hence, when the data was written into the data channel, the OS added a pointer to the data channel in the send queue. When the send queue was processed, the data from the data channel was packed into a data packet which was then written into the FPGA send buffer. From there, it was transferred by the transfer function to the FPGA receive buffer to give an impression to the processor that a new data packet has arrived. The processor then unpacked the packet, wrote the data into the data channel and sent back an acknowledgement packet. The acknowledgement packet was written into the FPGA send buffer from where it was transferred to the FPGA receive buffer to give an impression to the processor that an acknowledgment packet for the data packet sent earlier has arrived. Thus, we were able to emulate inter-processor communication between two processes.

Emulate 3-phase boost rectifier application
In order to emulate a 3-phase boost rectifier application, we again modified the dataflow descriptor file. The ADC_DC driver and the 1-D regulator ECOs were assigned to one processor while the remaining ECOs were assigned to another processor. The protocol was then allowed to execute normally with only the transfer function transferring packet data and acknowledgement packet from FPGA send buffer to the FPGA receive buffer.

Implementing the protocol on two universal controllers for a two-process application
Although we wanted to implement the protocol on two universal controllers, only one universal controller was ready at the time of writing this thesis. Hence, we connected a single processor to itself and then implemented the protocol for a two-process application. Again, we modified the dataflow descriptor file to give an impression that the two communicating processes were on different processors.

Although the protocol was implemented successfully, we were not able to capture timing information. This was so because the FPGA was implemented such that the processor could send only one packet per switching cycle. As a result, the processor after sending a data packet in one switching cycle, was able send the acknowledgement packet only in the next switching cycle.

Hence, we were not able to gather accurate timing information for a complete cycle of the protocol.

Implementing the protocol on two universal controllers for a boost rectifier application
In the dataflow descriptor file, the ADC_DC driver and 1-D regulator driver ECOs were assigned to one processor while the remaining ECOs were assigned to another processor. The ADC_DC driver was fired using interrupt-driven data channel. The 1-D regulator ECO regulated the dc voltage and generated the current loop reference d_{d_ref}. The current loop reference was then communicated to the current loop on another processor.

As only one universal controller was ready at the time of writing this thesis, we connected the single universal controller to itself and then implemented the protocol for the boost rectifier application. Again, we were not able to capture the timing information for a complete cycle of the protocol as the FPGA implementation allowed for only one packet to be sent per switching cycle.

5.2.3 Modifications to be made to the hardware to enable validation of the protocol

To allow for complete validation of the protocol, the universal controller hardware must be modified as follows:

- At the time of writing this thesis, only one universal controller was available. In future, with the availability of more universal controllers, we will be able to test the protocol using two or more universal controllers.

- Currently, the universal controller can send only one packet per switching cycle. Hence, it is not possible to accurately measure the time required for one complete cycle of the protocol. Modifications need to be made to the FPGA so that multiple packets can be sent per switching cycle.

- At this stage, there is no way in which the FPGA can indicate arrival of a new packet to the DSP. As a result, the DSP has to periodically check the FPGA receive buffer for the arrival of a new packet. Some mechanism such as an interrupt needs to be implemented so that the DSP can be made aware of the arrival of a new packet.

Chapter 6

Conclusions and Future Work

Multi-processor applications with configuration-specific code often require lot of code changes when the system is reconfigured. Also, applications designed for a single processor often require many code modifications to enable their use in multi-processor systems. This greatly affects the flexibility and ease of use of such applications. In this thesis, we have designed and implemented an inter-processor messaging protocol that addresses these issues. The protocol ensures both location transparency and distribution transparency, and provides fault tolerance against network failures. The protocol is implemented for power electronics control systems that have been designed using dataflow architecture. The protocol supports asynchronous communication to ensure efficient inter-processor communication between dataflow processes.

The thesis further includes an analysis of the network parameters to determine the time required for a complete cycle of the inter-processor communication.

In summary, the main points of this thesis are:

- The operating system handles the inter-processor communication, thereby allowing the application code to be written independently of the number of processors and/or process allocation strategy.

- An acknowledgement packet is sent to the source processor for every data packet received by the destination processor. This provides fault tolerance against network failures.

- The protocol allows for asynchronous communication. Hence, the source ECO is not blocked waiting for the destination ECO to be ready to receive the data.

- Using the analytical equation, it is possible to determine the delay involved in inter-processor communication. For many applications, this inter-processor communication delay may be an important factor in allocating of processes to processors.

6.1 Future Work

This thesis presents an inter-processor messaging protocol for communication between distributed processes. Since this was the first attempt at implementing the protocol hence simplicity was given preference over efficiency.

In the current version of the protocol, each packet contains message for only one data channel. In the future, work needs to be done to pack multiple messages with the same source and destination processors in a single packet.

This thesis does not include the validation of the network analysis as the hardware required for testing is not yet available. Future work will involve testing the protocol on the universal controllers, measuring the time required for one cycle of the protocol and then comparing it with the theoretically obtained value.

In the future, work will be done on implementing the protocol for controllers physically stacked together. The performance of the protocol will then be evaluated. As there won't be any delay over the ring, the performance is expected to be much better than the one observed over the ring.

Also, research is being done on PESNet to develop a set of communication protocols called the PESNet Interface (PESNI) that specify the physical, media-access and network layers of the ISI OSI reference model. The PESNI will consist of 3 specifications, physical layer protocol (PLP), link layer protocol (LLP) and network layer protocol (NTP). These will serve to standardize the protocol, provide improved fault tolerance and will also reduce the round trip time by the use of routing methods. It would be interesting to measure the performance of our messaging protocol on the improved version of PESNet.

References

[Guo02] J. Guo, S. Edwards, and D. Boroyevich. “Elementary control objects: Toward a dataflow architecture for power electronics control software," *Proc. IEEE 33rd Annual Power Electronics Specialists Conf.* (PESC 2001), June 2002.

[Francis01] J. Francis, and D. Boroyevich. “Design of a Universal Controller for Distributed Control and Power Electronics Applications.” CPES 2001 Power Electronics Seminar and NSF/Industry Annual Review, April, 2001.

[Francis02] J. Francis, J. Guo, and S. Edwards. “Protocol Design of Dual Ring PESNet (DRPESNet).” CPES 2002 Power Electronics Seminar and NSF/Industry Annual Review, April 2002.

[Singh02-1] K. Singh and S. Edwards. “DARK: Designing a high performance micro-kernel for power electronics controllers”. CPES 2002 Power Electronics Seminar and NSF/Industry Annual Review, April 2002.

[Singh02-2] K. Singh, *Design and Evaluation of an Embedded Real-time Micro-kernel,* master’s thesis, Dept. Computer Science, Virginia Polytechnic Inst. And State Univ., Blacksburg, VA, 2002

[Protic98] J. Protic, M. Tomasevic, and V. Milutinovic, “Distributed Shared Memory: Concepts and Systems,” IEEE Computer Society, Los Alamitos, Calif., 1998, pp. 9-41.

[Veldema01] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal, “Runtime optimizations for a Java DSM implementation,” *Proc. 2001 Joint ACM-ISCOPE conference on Java Grande*, Palo Alto, Calif., 2001, pp. 153-162.

[Hu03] Y.C. Hu, W. Yu, A. Cox, D. Wallach, and W. Zwaenepoel. “Run-time support for distributed sharing in safe languages,” *ACM Transactions on Computer Systems (TOCS)*, vol. 21, Issue 1, Feb. 2003, pp. 1-35.

[Bacellar98] L. Bacellar, and B. Upender, “A Dependable Distribution-Transparent Remote Method Invocation Model for Object-Oriented Distributed Embedded Computer Systems,” *Proc. 1st IEEE International Symposium on Object-Oriented Real-Time Distributed Computing* (ISORC’98), Kyoto, Japan, Apr. 1998, pp. 467-476.

[Schmidt98] D.C. Schmidt and S. Vinoski, "Introduction to CORBA Messaging," *C++ Report*, vol. 10, Nov./Dec. 1998.

[Arulanthu00] A.B. Arulanthu, C. O'Ryan, D.C. Schmidt, M. Kircher and J. Parsons, “The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging,” *Proc. IFIP/ACM International Conference on Distributed systems platforms,* N.Y., Apr. 2000, pp. 208-230.

[OMG99] Object Management Group, "The Common Object Request Broker: Architecture and Specification", 2.3 ed., June 1999.

[Sun96] Sun Microsystems, Inc. *Java Remote Method Invocation Specification,* 1996.

[Sun99] Sun Microsystems Inc. "Java Remote Method Invocation – Distributed Computing for Java", Nov. 1999.

[Ahuja00] S.P. Ahuja, and R. Quintao, "Performance evaluation of Java RMI: a distributed object architecture for Internet based applications", *Proc. 8th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (MASCOTS 2000), San Francisco, CA, August 2000.

[Williams03] Sara Williams and Charlie Kindel, "The Component Object Model: A Technical Overview," *Microsoft Corporation*, http://msdn.microsoft.com/library/ default.asp?url=/library/en-us/dncomg/html/msdn_comppr.asp (available 17th July, 2003).

[Kipfer99] P. Kipfer, and P. Slusallek. "Transparent Distributed Processing for Rendering," Parallel Visualization and Graphics Symposium (PVG)'99, 1999, pp. 39-46.

[Diaconescu02-1] R.E. Diaconescu, and R. Conradi. "A Data Parallel Programming model Based on Distributed Objects," *Proc. IEEE International Conference on Cluster Computing* (Cluster 2002), Sept. 2002.

[Diaconescu02-2] R. Diaconescu, "Distributed Component Architecture for Scientific Applications," *Proc. 40th International Conference on Technology of Object-Oriented Languages and Systems* (TOOLS Pacific 2002), Feb. 2002

[Diaconescu02-3] R.E. Diaconescu, *Object Based Concurrency for Data Parallel Applications: Programmability and Effectiveness*, doctoral dissertation, Dept. Computer Science, Norwegian Univ. of Science and Tech., Trondheim, Norway, 2002.

[Haridi99] S. Haridi, P.V. Roy, P. Brand, M. Mehl, R. Scheidhauer, and G. Smolka. "Efficient Logic Variables for Distributed Computing," ACM Transactions on Programming Languages and Systems (TOPLAS), 1999.

[Schollmeyer91] Schollmeyer, M "Linda and parallel computing-running efficiently on parallel time," potentials IEEE, Vol. 10, Issue 3, Oct 1991, pp. 43-45.

[Gelernter85] David Gelernter, "Generative Communication in Linda," ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 7 Issue 1, Jan 1985.

Appendix A

Code listing

```
/*=============================================================================*\
| pesnet_send.h
*-----------------------------------------------------------------------------*
| This file checks for space in the FPGA send buffer. If there is space in the buffer,
| then it will call the read_bytes_packet macro to prepare the data packet.
\*=============================================================================*/

#include <stdio.h>
#include "OS_cfg.h"

#define pesnet_send_handle()                                                    \
{                                                                               \
        QueuePointer qptr;                                                      \
        while (sendq_front != sendq_rear)                                       \
        {                                                                       \
                if(((*ctrl_reg_1) & 1) == 0)                                    \
                {                                                               \
                        qptr=send_queue.sendq[sendq_front];                     \
                        mod_plus_1(sendq_front, SENDQ_LENGTH);                  \
                        read_bytes_packet(qptr);                                \
                }                                                               \
        }                                                                       \
}
```

```
/*=============================================================================*\
| pesnet_receive.h
*-----------------------------------------------------------------------------*
| This file checks if the packet received by the FPGA is a data packet or ack packet. For
| a data packet, it calls the write_bytes_packet macro to write data into the data channel.
| For ack packet, it calls the delete_ack_bytes macro to delete data from data channel
| based on number of bytes acknowledged.
\*=============================================================================*/
#include <stdio.h>
#include "OS_cfg.h"

#define pesnet_receive_handle()                                                 \
{                                                                               \
        QueuePointer qptr;                                                      \
        int code = 1;                                                           \
        Combined_Packet receive;                                                \
        int channel_id;                                                         \
                                                                                \
        if(code == 1)                                                           \
                receive = receiveA_packet;                                      \
        else                                                                    \
                receive = receiveB_packet;                                      \
        if (((*(receive.cmd_data)) & 0xf0000000)== data_packet)                 \
        {                                                                       \
                channel_id = ((*(receive.cmd_data)) & 0xFF0000) >> 16;          \
                qptr = (QueuePointer)OS_All_DC[channel_id];                     \
                if(qptr->array_dimensions == null)                              \
                        write_bytes_packet (qptr, receive);                     \
                else                                                            \
                        write_raw_packet(qptr, receive);                        \
                (*ctrl_reg_1) |= 1;                                             \
        }                                                                       \
        else                                                                    \
        {                                                                       \
                int channel_id, ack_bytes;                                      \
                                                                                \
                channel_id=((*(receive.cmd_data)) & 0xFF0000) >> 16;            \
                ack_bytes = ((*(receive.cmd_data)) & 0x0F000000) >> 24;         \
                qptr=(QueuePointer)OS_All_DC[channel_id];                       \
                                                                                \
                delete_ack_bytes(qptr, ack_bytes);                              \
        }                                                                       \
}
```

```
/*=============================================================================*\
| read_bytes_packet (qptr)
*-----------------------------------------------------------------------------*
| qptr    is a pointer to a data channel which has data to be sent across the network
| This macro packs data from data channel into a data packet.
\*=============================================================================*/
#if (USE_MAILBOX)
#define read_bytes_packet(qptr)                                                 \
{                                                                               \
        int i;                                                                  \
                                                                                \
        log ( "read_typed_object: transferring data\n" );                       \
        for (i=0;i<qptr->element_size;i++)                                      \
                *(send_packet.data[i]) = *(qptr->buffer + qptr->front + i);     \
        *(send_packet.cmd_data) = data_packet | proc1_shift | (qptr->element_size << 24)\
                | (qptr->channel_id << 16) | DFG_Node_Set[DFG_Edge_Set[qptr->   \
                channel_id].sink].assigned_processor;                           \
                                                                                \
        qptr->allocation_type=sent;                                             \
        ack_queue[qptr->channel_id]=true;                                       \
}
#else
#define read_bytes_packet(qptr)                                                 \
{                                                                               \
        log ( "read_typed_object: transferring data\n" );                       \
                                                                                \
        if (((qptr->num_entries) * (qptr->element_size)) == 1)                  \
        {                                                                       \
                *(send_packet.data[0]) = *(qptr->buffer + qptr->front);         \
                *(send_packet.cmd_data) = data_packet | proc1_shift | (0x1000000)\
                        | (qptr->channel_id << 16) |                            \
                        DFG_Node_Set[DFG_Edge_Set[qptr-                         \
                        >channel_id].sink].assigned_processor;                  \
        }                                                                       \
        else                                                                    \
        {                                                                       \
                unsigned int rem_capacity;                                      \
                int size;                                                       \
                int local_front;                                                \
                int pack_bytes;                                                 \
                                                                                \
                local_front = qptr->front;                                      \
                size = ((qptr->num_entries)*(qptr->element_size));              \
                if (size >= PACKET_SIZE)                                        \
                        pack_bytes = (PACKET_SIZE/qptr->element_size)*          \
                                        (qptr->element_size);                   \
```

```
		else                                                                    \
			pack_bytes = size;                                              \
                                                                                \
		*(send_packet.cmd_data) = data_packet | proc1_shift | (pack_bytes << 24) \
			| (qptr->channel_id << 16) |                                    \
			DFG_Node_Set[DFG_Edge_Set[qptr-                                 \
			>channel_id].sink].assigned_processor;                          \
		rem_capacity = qptr->size_in_bytes - local_front;                       \
		if (rem_capacity >= pack_bytes)                                         \
		{                                                                       \
			int i;                                                          \
			for (i=0;i<pack_bytes;i++)                                      \
			{                                                               \
				*(send_packet.data[i]) = *(qptr->buffer+local_front+i);  \
			}                                                               \
		}                                                                       \
		else                                                                    \
		{                                                                       \
			int i;                                                          \
			for (i=0; i < rem_capacity; i++)                                \
			{                                                               \
				*(send_packet.data[i]) = *(qptr->buffer+local_front+i);  \
			}                                                               \
			for (i=0; i < (pack_bytes-rem_capacity); i++)                   \
			{                                                               \
				*(send_packet.data[i + rem_capacity]) = *(qptr->buffer+i); \
			}                                                               \
		}                                                                       \
	}                                                                               \
	qptr->allocation_type = sent;                                                   \
	ack_queue[qptr->channel_id] = true;                                             \
}
#endif
```

```
/*=============================================================================*\
| PACKET_BLOCK_ON_MAILBOX (qptr, receive)
*-----------------------------------------------------------------------------*
| qptr          is a pointer to a data channel to which data from data channel is to be
|               written
| receive       is the pointer to the FPGA receive buffer which contains the data packet
|               received from the source processor
| This macro writes data from data packet into mailbox data channels and then prepares
| acknowledgement packet, acknowledging the number of bytes written into the data
| channel.
\*=============================================================================*/

#if (DYNSCHD)
#define PACKET_BLOCK_ON_MAILBOX (qptr, receive)                                 \
{                                                                               \
        if ((qptr->overflow_style == OS_Block ) && (full(qptr)))                \
        {                                                                       \
                *(packet.cmd_data) = ack_packet | (proc2_shift) |               \
                                     (qptr->channel_id << 16)                   \
                                     | (((*(receive.cmd_data)) & 0xFF00) >> 8); \
        }                                                                       \
        else                                                                    \
        {                                                                       \
                int i;                                                          \
                for (i=0; i < qptr->element_size; i++)                          \
                        *(qptr->buffer+i)  = receive.data[i];                   \
                qptr->num_entries = 1;                                          \
                *(packet.cmd_data) = ack_packet | (proc2_shift) | (0x1000000) | \
                                     (qptr->channel_id << 16) |                 \
                                     (((receive.cmd_data) & 0xFF00) >> 8);      \
                setmask_on_write( qptr );                                       \
        }                                                                       \
}
#else
#define PACKET_BLOCK_ON_MAILBOX (qptr, receive)                                 \
{                                                                               \
        int i;                                                                  \
        for (i=0; i < qptr->element_size; i++)                                  \
                *(qptr->buffer+i)  = receive.data[i];                           \
        qptr->num_entries = 1;                                                  \
        *(packet.cmd_data) = ack_data | (proc2_shift) | (0x1000000) |           \
                             (qptr->channel_id << 16) |                         \
                             ((receive.cmd_data) & 0xFF00) >> 8);               \
        setmask_on_write( qptr );                                               \
}
#endif
```

```
/*=============================================================================*\
| PACKET_BLOCK_ON_WRITE (qptr, packet_bytes, avail_bytes, OW_Newest)
*-----------------------------------------------------------------------------*
| qptr          is a pointer to a data channel to which the data from the data packet is to
|               be written
| packet_bytes  indicates the number of data bytes that are sent in the data channel
| avail_bytes   indicates the available bytes in the data channel
| OW_Newest     will be set to 1 by the macro if the overflow style of the data channel is to
|               overwrite the newest element
| This macro checks if there is space in the data channel for the packet data. If there is
| insufficient space, then it will make space for the packet data if the overflow policy is
| to overwrite the oldest or newest element. The macro also updates the num_entries field
| based on the data bytes that are going to be written into the data channel
\*=============================================================================*/
#if (DYNSCHD)
#define PACKET_BLOCK_ON_WRITE ( qptr, packet_bytes, avail_bytes, OW_Newest )\
{                                                                             \
        int length;                                                           \
        length = packet_bytes-avail_bytes;                                    \
        if (length > 0)                                                       \
        {                                                                     \
                switch ( qptr->overflow_style )                               \
                {                                                             \
                        case OS_Block:                                        \
                                packet_bytes = avail_bytes;                   \
                                break;                                        \
                        case OS_Overwrite_Newest:                             \
                                if (qptr->front == qptr->rear)                \
                                {                                             \
                                        qptr->rear -= qptr->element_size;     \
                                        if ( qptr->rear < 0 )                 \
                                        {                                     \
                                                qptr->rear +=                 \
                                                qptr->size_in_bytes;          \
                                        }                                     \
                                }                                             \
                                OW_Newest = 1;                                \
                                break;                                        \
                        case OS_Overwrite_Oldest:                             \
                                qptr->front += length;                        \
                                if (qptr->front >= qptr->size_in_bytes)       \
                                        qptr->front=qptr->front - qptr->size_in_bytes;\
                                        break;                                \
                }                                                             \
                qptr->num_entries = qptr->size_in_elts;                       \
        }                                                                     \
```

```
        else                                                                    \
                qptr->num_entries += (packet_bytes/qptr->element_size);         \
}
#else
#define PACKET_BLOCK_ON_WRITE( qptr, packet_bytes, avail_bytes, OW_Newest )\
{                                                                               \
        qptr->num_entries += (packet_bytes/qptr->element_size);                 \
}
#endif
```

```
/*=============================================================================*\
| write_bytes_packet (qptr, receive)
*-----------------------------------------------------------------------------*
| qptr          is a pointer to a data channel to which the data from the packet is to be
|               written
| receive       is the pointer to the FPGA receive buffer which contains the data packet
|               received from the source processor
| This macro writes data from data packet into data channel and then prepares the
| acknowledgement packet based on number of bytes acknowledged.
\*=============================================================================*/
#if (USE_MAILBOX)
#define write_bytes_packet(qptr, receive)                                       \
{                                                                               \
        log("Writing to mailbox\n");                                            \
        PACKET_BLOCK_ON_MAILBOX(qptr, receive, packet_data)                     \
}
#else
#define write_bytes_packet(qptr, receive)                                       \
{                                                                               \
        int avail_bytes;                                                        \
                                                                                \
        avail_bytes = qptr->size_in_bytes - ((qptr->num_entries)*(qptr->element_size));  \
        if ((avail_bytes==0) && (qptr->overflow_style == OS_Block))             \
        {                                                                       \
                *(send_packet.cmd_data) =   ack_packet | (proc2_shift) |        \
                                            (qptr->channel_id << 16)            \
                                            | (((*(receive.cmd_data)) & 0xFF00) >> 8);  \
        }                                                                       \
        else                                                                    \
        {                                                                       \
                int local_rear;                                                 \
                int packet_bytes;                                               \
                int OW_Newest = 0;                                              \

                packet_bytes = ((*(receive.cmd_data)) & 0x0F000000) >> 24;      \
                                                                                \
                PACKET_BLOCK_ON_WRITE( qptr, packet_bytes, avail_bytes, OW_Newest )\
                                                                                \
                *(send_packet.cmd_data) =   ack_packet | (proc2_shift) |        \
                                            (packet_bytes << 24) |              \
                                            (qptr->channel_id << 16) |          \
                                            (((*(receive.cmd_data)) & 0xFF00) >> 8);  \
                local_rear=qptr->rear;                                          \
                                                                                \
                if (packet_bytes == 1)                                          \
                {                                                               \
```

```
        *(qptr->buffer + local_rear) = *(receive.data[0]);                  \
        local_rear++;                                                       \
        if (local_rear == qptr->size_in_bytes) local_rear = 0;              \
}                                                                           \
else                                                                        \
{                                                                           \
        int qptr_bytes;                                                     \
        qptr_bytes = qptr->size_in_bytes;                                   \
        if (OW_Newest == 1)                                                 \
        {                                                                   \
                int *src;                                                   \
                int *src_final;                                             \
                src = receive.data[0] + packet_bytes - qptr->element_size;  \
                src_final = receive.data[0] + packet_bytes;                 \
                while (src != src_final)                                    \
                {                                                           \
                        *(qptr->buffer + local_rear) = *(src++);            \
                        local_rear++;                                       \
                        if ( local_rear == qptr_bytes)                      \
                                local_rear = 0;                             \
                }                                                           \
        }                                                                   \
        else                                                                \
        {                                                                   \
                char* dest;                                                 \
                int* src;                                                   \
                char* dest_final;                                           \
                int* src_final;                                             \
                                                                            \
                dest = (qptr->buffer + local_rear);                         \
                src = receive.data[0];                                      \
                src_final = receive.data[0] + packet_bytes;                 \
                dest_final = qptr->buffer + qptr_bytes;                     \
                while (src != src_final)                                    \
                {                                                           \
                        *(dest++) = *(src++);                               \
                        if (dest == dest_final)                             \
                                break;                                      \
                }                                                           \
                dest = (qptr->buffer);                                      \
                while (src != src_final)                                    \
                {                                                           \
                        *(dest++) = *(src++);                               \
                }                                                           \
                local_rear += packet_bytes;                                 \
                if (local_rear >= qptr_bytes)                               \
```

```
                        local_rear -= qptr_bytes;                                   \
                    }                                                               \
                }                                                                   \
                qptr->rear = local_rear;                                            \
                                                                                    \
                setmask_on_write (qptr);                                            \
        }                                                                           \
}
#endif
```

```
/*=============================================================================*\
| delete_ack_bytes (qptr, num_of_bytes)
*-----------------------------------------------------------------------------*
| qptr                  is a pointer to a data channel identified by the acknowledgement
|                       packet
| num_of_bytes          is the number of data bytes acknowledged
| This macro deletes data from the data channel based on the number of bytes
| acknowledged.
\*=============================================================================*/
#if (USE_MAILBOX)
#define delete_ack_bytes( qptr, num_of_bytes)                                   \
{                                                                               \
        ack_queue[qptr->channel_id] = false;                                    \
        if (qptr->difference != qptr->element_size)                             \
        {                                                                       \
                qptr->num_entries = 0;                                          \
                (qptr->sink_process)->in_ports_ready &=                         \
                        ~( 1 << DFG_Edge_Set[qptr->index].sink_in_port);        \
                FREE_SOURCE_PROCESS( qptr );                                    \
        }                                                                       \
        else                                                                    \
        {                                                                       \
                qptr->allocation_type=waiting_to_send;                          \
                insert_send_queue(qptr);                                        \
                mod_plus_1(send_queue.rear, MAX_SEND_QUEUE_LENGTH);             \
                qptr->difference = 0;                                           \
        }                                                                       \
}
#else
#define delete_ack_bytes (qptr, num_of_bytes)                                   \
{                                                                               \
        int size_in_bytes;                                                      \
        size_in_bytes = qptr->size_in_bytes;                                    \
                                                                                \
        if ((qptr->overflow_style == OS_Block) || (qptr->difference==0))        \
        {                                                                       \
                qptr->front += num_of_bytes;                                    \
                qptr->num_entries-=(num_of_bytes/qptr->element_size);           \
                if (qptr->num_entries==0)                                       \
                        qptr->allocation_type=empty;                            \
        }                                                                       \
        else if (qptr->overflow_style == OS_Overwrite_Newest)                   \
        {                                                                       \
                if (num_of_bytes <= (size_in_bytes - qptr->difference))         \
                {                                                               \
                        qptr->front += num_of_bytes;                            \
```

```
                qptr->num_entries-=(num_of_bytes/qptr->element_size);          \
        }                                                                      \
        else                                                                   \
        {                                                                      \
                qptr->front += size_in_bytes - qptr->difference;               \
                qptr->num_entries -= ((size_in_bytes - qptr->difference)/qptr->  \
                                        element_size);                         \
        }                                                                      \
        qptr->difference = 0;                                                  \
    }                                                                          \
    else                                                                       \
    {                                                                          \
        if (num_of_bytes > qptr->difference)                                   \
        {                                                                      \
                qptr->front += (num_of_bytes - qptr->difference);              \
                qptr->num_entries -= ((num_of_bytes - qptr->difference)/qptr->  \
                                        element_size);                         \
                qptr->difference = 0;                                          \
        }                                                                      \
        else                                                                   \
        {                                                                      \
                qptr->difference = 0;                                          \
                ack_queue[qptr->channel_id]=false;                             \
                return;                                                        \
         }                                                                     \
    }                                                                          \
    if ( qptr->front >= (size_in_bytes))                                       \
        qptr->front = qptr->front - size_in_bytes;                             \
                                                                               \
    ack_queue[qptr->channel_id]=false;                                         \
    FREE_SOURCE_PROCESS (qptr);                                                \
}
#endif
```

```
/*=============================================================================*\
| Code to set up the priority and allocation_type field values for distributed data channels
*-----------------------------------------------------------------------------*
| This code sets up the priority of the distributed data channel as the highest priority of
| firing rule of the sink process associated with the data channel.
\*=============================================================================*/
for (i=0; i<NUM_DFG_EDGES; i++)
{
        DC_ptr = (DC_Queue *)OS_All_DC[i];
        DC_ptr->source_process = OS_All_Process[DFG_Edge_Set[i].source];
        DC_ptr->sink_process = OS_All_Process[DFG_Edge_Set[i].sink];

        if (DFG_Node_Set[DFG_Edge_Set[i].source].assigned_processor ==
                DFG_Node_Set[DFG_Edge_Set[i].sink].assigned_processor)
        {
                DC_ptr->allocation_type = null;
                DC_ptr->dist_priority = 0;
        }
        else
        {
                int max_priority = 0;
                Firing_Rule  firing_rule_ptr;
                unsigned int mask;

                sink_process = DC_ptr->sink_process;
                firing_rule_ptr = DFG_Node_Set[DFG_Edge_Set[i].sink].firing_rules[0];

                while (!FIRING_MASK_TERMINATED(firing_rule_ptr))
                {
                        mask = 0x1 << DFG_Edge_Set[i].sink_in_port;
                        if (((firing_rule_ptr->mask) & mask) == mask)
                        {
                                if (max_priority < firing_rule_ptr->priority)
                                {
                                        max_priority = firing_rule_ptr->priority;
                                }
                        }
                        (firing_rule_ptr)++;
                }

                DC_ptr->dist_priority = max_priority;
                DC_ptr->allocation_type=empty;
        }
        NUM_DISTR_EDGES++;
}
```

Vita

Parool Mody was born in Gujarat, India on February 26, 1978. She did her Bachelor degree in Computer Engineering from University of Mumbai, India. As part of her undergraduate degree, she developed an internet firewall for network security. She worked for 2 years as a Systems Analyst at Tata Infotech Ltd, a software company in India.

She joined Virginia Polytechnic Institute and State University in August 2001 for pursuing Masters degree in Computer Science. She will be graduating in August 2003. She is currently working as a QA Analyst at Integrated Software Solutions.