A Reconfigurable Distributed Process Control Environment for a Network of PC's Using Ada and NetBIOS.

by
Mark Charles Randelhoff
(B.Sc.Eng.)

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Department of Electronic Engineering, University of Natal, South Africa.

Durban

February 1992.

For my Mother,
and my Father,
thank you for all your love
through the years

Acknowledgements

I give thanks to God and his Son, Jesus Christ, for giving meaning to my life no matter how often I stray.

With regard to this thesis, I would firstly like to acknowledge the tremendous support provided by my family.

I would like to thank my supervisor Mr D.C. Levy for making this wonderful oppertunity to explore possible, for allowing me the freedom to choose an independent path of research.

Thank you to Dr. Joyce Tokar, for all the advice and time she has spent helping me. Thank you for your incredible patience.

The many discussions and contributions of my friend Richard Scott, at times when I was at wits end he often provided a catalyst for new ideas.

The support provided by Mr Roger Peplow, Mr Rob Brain and Mr Alan Barrett in maintaining the computing resources is much appreciated.

The rest of the Postgrads and Staff for having my sojourn an enjoyable and fulfilling one.

Finally the financial support of my bursars Sasol Limited, who have sponsored me throughout my undergraduate and postgraduate studies, and to the University of Natal.

TABLE OF CONTENTS

CHAPTER ONE

Introduction

1.1 OBJECTIVES OF THE SYSTEM DESIGN

This objective of this project was to study the use of Ada[1] in embedded real-time control systems (DERTCS) that are distributed across low cost personal computer local area networks (PC LANS). As a result a communications subsystem was developed to manage a distributed set of tasks in such a way that the system may be reconfigured without shutdown. This subsystem is intended for use by a process control engineer in implementing a control system for a small industrial plant, cheaply, in real-time, while taking advantage of the tools available today.

PCs are easily and cheaply obtainable. They are finding wide use in real-time control systems[2] where many suppliers such as Burr Brown, Data Translation, and Metrabyte produce add on hardware in the form of A/D and D/A converters, strain gauge and thermocouple interfaces etc.

Distributed systems are also finding wide application in real-time systems as they are robust, are scalable and facilitate the placement of CPU power close to the plant.

The programming language Ada was developed by the United States Department of Defense specifically for embedded real-time control systems. Ada is designed to be a robust, high level language, with support for strong typing, exception handling, tasking, packages and data abstraction.

The availability of Ada on PCs, complete with support for the real-time tasking mechanism in the language, has made Ada an effective choice for implementing cheap and reliable real-time control systems.

In the eight years since the publication of the Ada standard, a number of deficiencies in the language have been recognised. This has led to the Ada9X effort[3] to refine the language definition and this is briefly discussed in a later chapter.

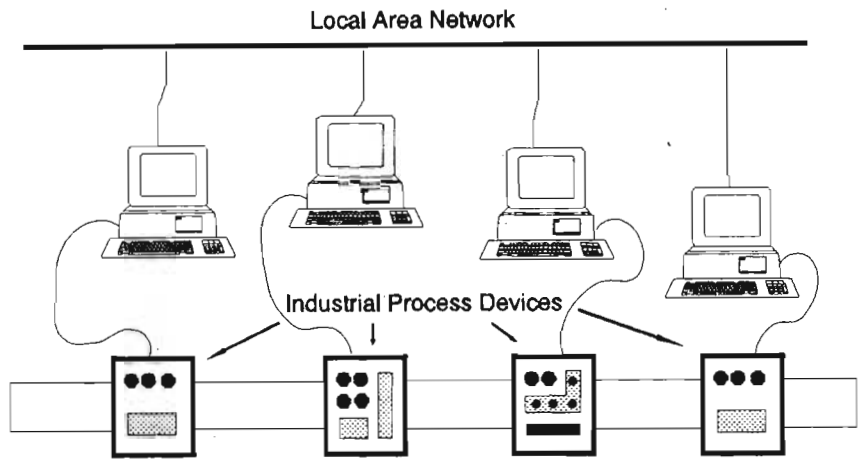

Figure 1.1 Distributed Process Control With PCs

This thesis briefly describes models and protocols for distributed control using Ada and a local area network of PCs, real-time systems, and some of the work on distributed Ada performed elsewhere. It goes on to describe the design and implementation of a communications package in Ada using an appropriate communications protocol. This package is expected to allow a process control engineer to implement a control philosophy easily and at low cost and must permit the system to be reconfigured while on line to allow the addition and removal of processors.

Two small distributed applications implemented using the system are described in order to show the viability of the design.

1.2 AN OVERVIEW OF THE SYSTEM

To make use of the communications system the process control engineer is expected to decompose an industrial process into a number of physical nodes, each of which is allocated a processor. The engineer identifies the functions performed at the node and specifies the data that must be transferred between these nodes.

The engineer writes controlling software (applications) for each function and the necessary calls to the communication library for the inter-processor communication. The application exists at

the application layer of the ISO model. The communications harness is placed within the presentation layer and it accesses the underlying network protocol, NetBIOS, which operates at the session layer.

The communications interface provides the process control application with facilities for reliably transferring information from node to node across the network. A set of functions are provided for instantiating and closing communications channels and then sending and receiving messages.

The system is designed to be reconfigurable; it allows the insertion of new physical nodes, without disengaging the controller from the process. This could be necessitated by a hardware failure or simply to enable the processor to be serviced. Similarly, where the overall process control strategy allows, and the process control engineer has provided the facility, the communications environment allows the removal of a remote node.

This chapter has briefly introduced the concept of Ada and given an outline of the design of a distributed Ada system. The following chapter will consider distributed systems, the problems associated with real-time systems, the suitability of the programming language Ada for systems of this type, and previous work in the field of distributed Ada.

CHAPTER TWO

Components of the System

2.0 INTRODUCTION

This chapter will discuss distributed systems and then Ada programming language. A further section addresses a number of projects that have studied various aspects of distributed Ada in the last few years. The implications and requirements of designing a real-time communication system for process control are considered. This chapter will conclude with a brief discussion on local area networks.

2.1 DISTRIBUTED SYSTEMS

Process control systems have often been implemented on uniprocessors where the controlling software is implemented in the form of multiple tasks. When a particular computer configuration is no longer capable of performing new or expanded applications the designers have two basic choices, namely to obtain a more powerful computer, or to add processors to the existing configuration. There is a growing tendency for modern computer hardware configurations for large, complex systems to use parallel processing. On such systems complex problems are divided into a number of smaller tasks and distributed amongst the processors available.

A distinction may drawn between multicomputer distributed systems consisting of two or more separate computers and distributed multiprocessor computer systems which normally have a large number of processors within a single computer. Multiprocessor computer systems are used primarily for numerically intensive computing in scientific applications although they are also used in real-time applications as host computers or for special purpose engines such as for graphical processing. Some authors (Sloman and Kramer [2], Enslow [5]) are of the view that a true distributed system must contain at least two separate computers. Multiprocessor computer systems will not be discussed here since the principles applicable to multicomputer systems are frequently applicable to multiprocessor systems.

2.1.1 Advantages and Disadvantages of Distribution.

The two main stimuli for the increased study of distributed systems are the technological changes that have taken place in the microchip industry, and the changing demands of the computer users.

The improved performance of the microchips has resulted in a price-performance ratio that favours the use of a number of lower performance processors rather than a single, more expensive, higher performance processor. The interconnection and communication costs between processors have fallen, and there are a number of local area networks, buses, and packet switched networks that are cost effective and easily available. The general growth in the use of computing has led to demands for improved performance and reliability, at a cheaper price. The potential benefits of distribution are numerous [2,4]:

Reduced Costs: In process control plants there can be a large saving of wiring costs by placing local processors adjacent to the devices that need to be controlled. Since wiring can often account for a major portion of the cost in a process control system the replacement of large quantities of parallel wiring with serial twisted pair or coaxial cable can reduce costs. In many applications this also may reduce the communication costs.

Modularity and Simpler Software: Distributed systems must be constructed in a modular fashion, where each component provides well defined interfaces or services to the rest of the system. This leads to simpler system design, installation and maintenance, and allows analytical verification techniques to be used. Functions in a process control plant may be easily identified and associated with physical nodes. These allocations may be made for optimum performance and easier maintenance results in an extended life cycle.

Concurrency Modelling. It is reasonably easy to construct an accurate model to correspond to the asynchronous processing that is inherent in most real-time process control systems.

Flexibility: Modularity which results in clearly defined interfaces, facilitates modification and extension of a system. Hardware flexibility is provided through replacement of processing

elements. Furthermore, it is possible to start with a small system and enlarge it by providing extra processing units to implement additional functions without a major redesign of the whole system. This also provides for an increased life cycle.

Integrity: Distributed systems have the potential to continue operating if a failure occurs in the hardware or the software. Critical resources may be replicated so the failure of a single unit does not preclude the continued operation of the system. Loose coupling in distributed systems (no shared memory) can provide some protection against memory corruption from a rogue process. Faults are more easily isolated through functional partitioning and distribution since it is relatively easy to identify hardware faults to a given physical node and far more cost effective to replicate a microprocessor circuitboard than a whole minicomputer system. Physical distribution of the system affords some protection against natural disasters or sabotage.

Performance: Response time may be reduced if most processing is performed locally, near the controlled device, and distributed processors may reduce processing bottlenecks.

In some cases, the potential advantages of distribution are outweighed by its disadvantages:

More Powerful Capabilities: A large central computer may offer a wider range of services and software than is available for personal computers. Some programs are too large to run on small machines and must be run on systems with large amounts of memory.

Operating Costs: The cost and number of personnel required to support a system is less if the system is centralised. Management and provision of services for the computing site is simpler for a centralised system.

Working Environment: Staff often prefer to work in an environment where they may interact with colleagues with similar interests. It is also more difficult to enforce standards when staff are dispersed and many services may be duplicated. If strict control is not exercised, additional processing functions may be added locally without comprehension of the cost and implications of that function to the performance of the overall system.

Communication Subsystem: The complexity of the distributed communication subsystem is often underestimated and the advantage of a simpler operating system may be offset by the overheads required to provide reliable communication.

Some of the advantages of centralisation may be realised in a distributed system by implementing resource sharing, for example secondary storage. Alternatively, a central larger system may be connected to a distributed system, giving a hybrid implementation. The choice between distribution and centralisation is dependent upon the need of the particular plant, and the application to be run, which together will determine the processing and communication requirements.

2.1.2 Models of Distribution.

In formulating a model of distribution, most authors view the system as comprising of components which may each be distributed. For example Sloman and Kramer [2], distinguish hardware, system software and application software, each or all of which are distributed. In many cases however, the distinction between application software and system software is blurred; an example is that of embedded applications where device control and task scheduling is usually the domain of the system software. An alternative view of system software is that it controls access to the hardware. In this case a distinction is drawn between control and data. Thus a distributed system contains distributed (or decentralised) hardware, control and data. The extent to which these dimensions need be distributed, for the system to be described as distributed, is again open to debate.

Enslow's Model

Enslow's model, of 1978 [5], requires that all three dimensions (hardware, control and data) be fully decentralised for a system to be classified as fully distributed. A simplified form of Enslow's model is given in figure 2.1. The distribution is expected to be transparent and the user should be unaware of the multiple processors that exist in the system. In this model, systems which contain a single control unit do not qualify as distributed systems, even if they have multiple, specialised processors to perform specific tasks. The system control must be

provided by multiple control units, cooperating with one another. Data must be partitioned and/or replicated, each part with its own local directory.

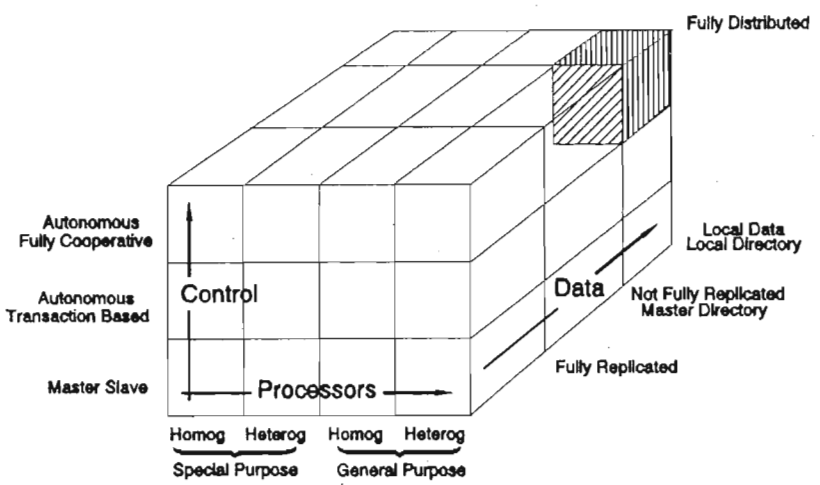

Figure 2.1 Enslow's Model for Distributed System Types

Taken From Sloman and Kramer figure 1.1

While this approach allows for a relative comparison of distribution, it is of more interest to identify those attributes that make a particular facet of a distributed system attractive, in particular for implementation in a control system.

Sloman and Kramer's Model

This model is more liberal in its definition. A system must have several autonomous processes and data stores and they must interact to achieve an overall goal. These processes exchange and coordinate their activities over a communications network. In particular, the system should support:

> *An arbitrary number of system and application processes.* The system should support the arbitrary increase and decrease of the number of processes to provide flexibility.
>
> *Modular physical architecture.* The system can consist of multiple interconnected and special purpose processing elements. These may be physically distributed and may alter in number during the systems lifetime.

Communication by message passing using a shared communication system. This provides for a transfer of messages in a cooperative, rather than a master slave manner. Communication based solely upon shared memory or shared address spaces is excluded.

System wide control. This is necessary to integrate the autonomous distributed processing elements into a single coherent system. The actual control depends upon the overall goal of the system.

This model does not allow for closely coupled systems that communicate through shared memory or crossbar switches to be considered distributed. Loosely coupled systems, capable of transferring files between one another are also not distributed unless they interact to achieve a common goal.

These respective models provide some perspective on distributed systems and the evolution toward a fully distributed system in the Enslow definition. Many applications exhibit some characteristics and are associated in current literature with distributed systems; distributed multi-microprocessor structures are an example of this.

2.1.3 Multi-Microprocessor Structures in Real-Time Control

We are concerned here with microprocessor structures for the control of real-time systems as well as the manipulation of data captured during real-time operations. Multi-microprocessor structures possess all of the advantages of distributed systems listed previously. Furthermore, rapid prototyping of a system is enhanced [4], since a portion of the overall design may be implemented cheaply and tested to reduce development time and cost, and eventually completely built on more expensive processors. The systems usually employ a set of heterogeneous processors that are connected via a LAN or through buses. Microprocessors with their low cost and wide range of performance characteristics allow application in a number or areas, for example data acquisition systems and automated process control systems [4].

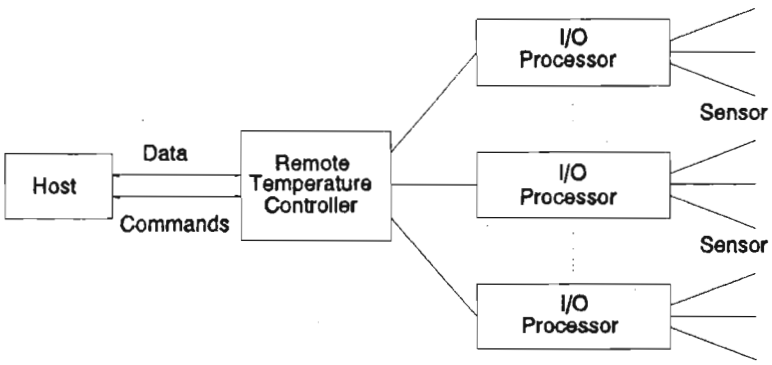

Figure 2.2 Remote Temperature Control System
Taken From Nielsen figure 3-12

Data Acquisition Systems:

In data acquisition systems, data from the plant being monitored is sampled by sensors which feeds the information to a host processor to allow it to make decisions and perform calculations. A typical example is a remote temperature sensor system used to control a set of furnaces. The sensors are driven by a set of microprocessors that send the data they receive to the control unit (remote temperature controller) which communicates with the host (figure 2.2).

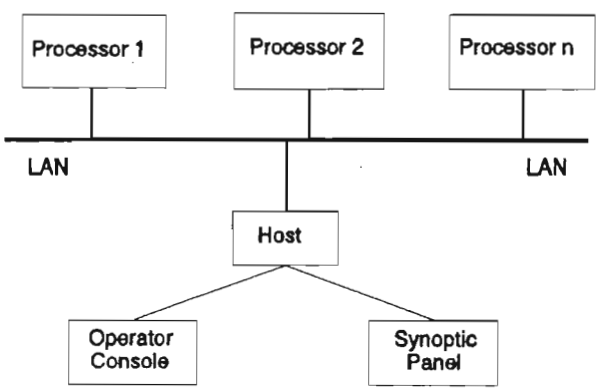

Figure 2.3 Bottling Plant Control System
Taken From Nielsen figure 3-13

Automated Process Control Systems: Automated process control systems acquire plant data, and use that data in a feedback configuration to control the plant. A typical example is a system which controls an assembly line in a bottling plant (figure 2.3). Data from each process device is obtained from an associated processor and relayed to the host for analysis. The host provides

a user interface for an operator to alter the set points and monitor the performance of the assembly line. This system facilitates implementation of fault tolerance by allowing redundant microprocessors to be included for any critical process device.

The performance of the overall system is dependent upon the communication system and the individual processing nodes. The effectiveness of personal computers as nodes in real-time applications is limited by the use of the MSDOS. Since MSDOS is not re-entrant this complicates pre-emption and the efficient use of interrupt handlers. Software running under MSDOS may perform real-time applications where the time constraints are sufficiently lax, such that the performance of the individual nodes of the system does not negatively impact upon the overall real-time system.

2.2 DISTRIBUTED ADA

Ada was initially intended for embedded programming and designed with several key goals in mind [6]:

Readability: It is recognised that programs are read far more often then they are written. It is important therefore to have a language that is easily read.

Strong Typing: Many errors are detected in compilation that would otherwise have led to run time errors.

Programming for Large Systems: Mechanisms for encapsulation, separate compilation and library management are necessary for writing any programs that are to be portable and of any size.

Exception Handling: This provides the ability for unexpected occurrences to be dealt with in a manner that will not compromise the integrity of the system.

Data Abstraction: Allows for extra portability and maintainability.

Tasking: Allows for several threads of control to exist within one program.

Generic Units: This is useful when the logic that is performed is independent of the type of values being manipulated.

Ada is well documented and supported. For the above reasons it is a logical choice as an implementation language. Several shortcomings been identified since its emergence, and the Ada community is attempting to address these issues in the second generation of Ada, Ada9X. In particular, Ada does not provide full support for distributed operation, with abstractions for communication over a network, in the form that languages specifically designed for distributed environments do. The Ada9X program is briefly discussed in a later chapter.

2.2.1 Ada in Distributed Systems

Ada does not provide explicit support for distribution in the manner of distributed programming languages such as MOD [7] since there are no abstractions for logical nodes or internode communication. Thus if one wishes to write Ada software for a distributed system, one will use the existing features of Ada, such as intertask message passing, and extend them. This problem has been addressed in different ways by a number of authors [8-12], but the two basic approaches are: one may write a single program and pass it through a pre-processor to map this program into several programs that may be individually compiled by a conventional compiler; or one might write several programs, one for each processor and implement the required communication software.

Furthermore, in both approaches the question of what granularity should be used for the unit of distribution must be addressed. Ada provides two natural choices for distribution: packages and tasks. Both approaches have their pros and cons [13]:

Tasks: Tasks are active units and their communication method, the rendezvous, may be used to describe the communication throughout the distributed network. Tasks may be defined as types and are permitted dynamic replication. Tasks may not be used as

library modules, although this limitation may be mitigated by library packages that export task types.

Packages: Packages are library units and thus lend themselves to easy reuse by different programs. The package's re-entrant interface allows for a higher degree of parallelism and Ada's designers considered packages to be the main unit of logical program decomposition [14].

The choice of Ada task or package units as the unit of distribution imposes restrictions on the use of the Ada constructs, the need for specialised development tools and difficulties in integration with existing run-time kernels. For example, the choice of tasks imposes a set of constraints on the form of Ada programs that will be possible to distribute. The model described by Arevelo and Alvarez [14] demands that these programs only allow definitions and task objects at their outermost level. Thus tasks on different nodes may interact only via messages and not through shared variables. Without memory sharing Volz[8] shows that both recursion and task termination may cause difficulties. Since packages do not possess their own thread of control, except that used for initialisation, they do not accurately model concurrent processes distributed across a loosely coupled network. Although these problems may be solved the overhead involved is significant and the recommendation is that tasks should be avoided where memory is tied to a single processor.

This has lead to other approaches for distribution within Ada. The virtual node approach has been adopted for several projects [9],[10],[11]. The major differences between the approaches lies in the varying ways in which they have represented the virtual node in Ada. More generally a virtual node may be considered to be a collection of library units, each with a communication interface to provide for communication between nodes. An application is designed as a single Ada program which is structured as a set of virtual nodes. These virtual nodes have most of the properties of an Ada main program. An abstraction for virtual nodes is provided and supported by the programming support environment. The usual method of communication is via a remote procedure call which involves exchanging a pair of messages. The call message carries the 'in' parameters and the identification for the service while the response carries the result of the request and the values for the 'out' parameters, and propagates any exceptions.

Tools transform the user-written program into one Ada program per processor, and also generate code stubs for inter-program communication. The programs and stubs are then compiled using a uniprocessor Ada compiler and linked to a modified run time library to produce a separate executable image for each processor.

This approach produces application programs that are stable against changes such as the number of processors in the target, or remapping between the virtual nodes and the processors [12]. Unfortunately, the approach makes the program structure heavily dependent upon the distributed nature of the hardware and it is no longer possible to base the structure of the program on functional considerations alone. This increases the complexity of the application software. Restrictions are placed upon the construction of the code to be transformed. For example the passing of access values and tasks must be forbidden since they have no meaning to the transformed source code. It is also extremely difficult to implement some rules of the language correctly. Examples are the semantics of the abort statement where a task to be aborted has remote dependents, exception propagation between nodes, and forcing multiple programs to follow an elaboration order that is implicit in the single program version.

The Distributed Ada Real-Time Kernel (DARK) [15,16] is a good example of distribution of multiple programs in Ada. The purpose of DARK was to provide a short term solution for programming distributed applications in Ada, with a view to a final solution being offered by the next generation of Ada, Ada9X. The DARK designers contend that the Ada tasking paradigm and view of time is immature, and unsuitable for real-time applications. The application is written as a number of Ada programs, which use DARK primitives, that are low level in nature. Several program restrictions are imposed, the most serious is that no Ada tasking primitives may be used. DARK suffers from the disadvantages characteristic of multiple program approaches. The structure of the application program is burdened by the knowledge of the hardware configuration. Changes in the hardware may then cause costly redesign of the system.

2.2.2 Classification of existing projects.

Existing projects have approached the distribution of Ada from a wide variety of directions. The type of hardware, requirements for fault tolerance and the constraints on compiler development are all factors to be considered. Bishop and Hasling [17] have developed a taxonomy whereby they classify distributed Ada projects according to four criteria:

Input - The system comprises a single program or multiple separate programs.
The units of partition.
The type of communication between units
The presence of configuration information.

Phase	Input	Communication	Partitions	Configuring	Examples
0	Multiple	Explicit	Restricted	Explicit	Transputer Ada
1	Single	Explicit	Restricted	Explicit	York, Diadem
2	Single	Implicit	Restricted	Explicit	Michigan,Mums NYU
3	Single	Implicit	Not Restricted	Explicit	Honeywell
4	Single	Implicit	Not Restricted	Implicit	None Yet

Figure 1.5 A Multiphase Classification of Distributed Ada
Taken From Bishop and Hasling [12]

As a result it is possible to group the projects in phases as shown in figure 1.5.

Phase 0: Multiple Instruction, Explicit Communications, Restricted Partitions, Explicit configuration.

The user is aware of the hardware configuration and writes a program for each physical node. The programs intercommunicate through a message passing interface that is supplied and included as a package. A frequent problem with this method is the lack of type checking across the individual programs. This may be avoided through the use of suitable library functions

written to alleviate the problem. Common type definitions may be kept in a library that is used for all the individual programs. The restrictions on Ada are numerous since there are no procedure calls, entry calls or global variables shared between the programs. This distributed system is easy to instantiate and off the shelf compilers may be used.

Phase 1: Single Instruction, Explicit Communications, Restricted Partitions, Explicit configuration.

The input here is a single program which has to be partitioned. In this case there may be more partitions, which are usually referred to as virtual nodes (discussed previously), than there are nodes. The designers of the system are required to specify what constitutes a partition. Ideally an Ada construct should be used, i.e. a task or package construct, but as discussed previously these constructs both have their limitations. It is necessary to identify what communication is permitted between partitions, how configuration information is communicated to the Ada system software, and how this program is translated to run on several processors.

Phase 2: Single Program, Implicit Communication, Restricted Partitions, Explicit Configuration.

At this stage there is no longer an explicit message passing system that the programmer may use. MUMS and NYU have adopted tasks as the unit of partition and the normal tasking and variable access methods for communication. These projects have altered the compiler to detect inter-node communication while Michigan Ada uses a sophisticated pre-processor to convert the single program into multiple programs communicating via a message passing system.

Phase 3: Single Instruction, Implicit Communication, No restrictions on partitioning, Explicit configuration.

The philosophy is simple - take a single existing program and couple it with a description on how it should be distributed. A further choice is either to make use of an intelligent compiler or to use a pre-processor and a conventional compiler. The problem reduces to the definition of the partition configuration language. An existing system to have achieved this is Honeywell's APPL language [12]. There are a number of deficiencies in this system which are further discussed by Bishop [17].

Phase 4: Single Program, Implicit Communication, No restrictions on partitioning and implicit configuration.
To date there is no working system at this level.

This thesis deals with a distributed system that falls very much within phase 0 of the above taxonomy.

2.2.3 Maintenance, Scalability and Reconfigurability of Phase 0

In some industries it is cost effective to keep production plants operating for all twenty four hours of the day. Major maintenance or equipment improvement occurs on an annual or bi-annual basis when the plants are shut down. When a portion of the plant halts during normal operation because of equipment failure, this may result in the whole production process coming to a halt such that the costs are enormous.

An advantage of the phase 0 model is that the understanding, debugging and on-line maintenance of a system is enhanced through a one to one mapping of a processor to a functional unit.
Furthermore, in real-time control it is desirable to understand the exact effect and cost that a particular channel or communication between two nodes may have on a system.

In the Phase 0 model failure of a single processor will not immediately cause the failure of the system as a whole, unless the actual industrial process cannot continue without this module. If the software at the other nodes permits the system to be maintained while the replacement of the failed node is achieved, the process may continue relatively undisturbed.

Furthermore, in a process control environment it is frequently desirable to add or remove a node as the industrial demands on the controller change. A multi-microprocessor system connected across a LAN can allow additional process units to be included while the rest of the process continues.

2.3 REAL-TIME SYSTEMS

The decrease in the cost of digital hardware has resulted in the widespread use of real-time systems to control commercial, industrial and communication systems. The requirements that are placed on these systems vary widely with their application and this has made it difficult to arrive at a definition for a real-time system. As in distributed systems, it is more convenient to fashion a definition from the properties that are commonly associated with such a system.

Real-time systems react in a way that affect the environment in which they operate [17] and may conveniently be divided into a controller and a controlled system [18]. In an industrial environment the controlled system will be the industrial process at hand while the computer system and its software is the controller. A number of software systems exist as a result of the differing constraints that are placed upon them by the systems that are controlled.

On-line Systems: The trend in modern business computer systems is toward fully on-line operation. The application is often run at times which do not make optimum use of the resources available because, for example, of transaction requests that are made from a collection of enquiry terminals only during business hours. The majority of real-time systems fall into this category.

Process-Control and Communications Systems: Process control systems include those for controlling chemical plants, missile systems and manufacturing machinery. Communication systems deal mainly with telephonic switching circuits and computer networks. Two aspects are immediately obvious; the systems must be extremely reliable and are time critical.

The terms *time critical* and consequently *real-time* have to be defined within the context of the system to be controlled. If a chemical process is being controlled then the time constant of the reaction defines what is considered time critical. Typical time deadlines are seconds to minutes in a chemical plant, seconds in a steel mill and milliseconds in a missile guidance system.

2.3.1 General Properties of Real-Time Systems

As mentioned before the general properties of real-time systems provide a convenient way of describing them and understanding their requirements. These properties are discussed in depth by a number of authors whose terminology and approach differ, although their content is essentially the same [4],[18],[19],[20].

Responsiveness to Environment: A real-time system must be capable of responding timeously to the changing climate of its environment, such that corrective action may be taken while it is still valid. The term *timeously* is as nebulous a term as *time critical* and requires supplementary definitions as shown above. This requirement has a direct effect upon the type of scheduling used in the run-time system for a real-time system with various threads of control. For example, a scheduler that implements time slicing whose time windows are too long for the process it controls will produce a sluggish controller and a poorly performing system.

Correctness and Completeness: A real-time system may frequently control systems that have the potential to do great physical or financial harm and must be carefully and rigorously tested.

Reliability: The system must provide a service that will guarantee a maximum failure rate and case degrade gracefully with failures.

Economy: If an on-line system, with more relaxed specifications, can provide a service which is cheaper to produce, run and maintain than a system with a faster response time, this system will be used in preference. Thus the economic cost must be considered when assessing the value of the above points.

In summary, real-time systems are controlled by software that must allow continuous operation, with stringent performance requirements and with facilities that allow for error detection and recovery. When this is distributed over a multicomputer system the problem is intensified in the sense that a communication system must exist such that it offers a reliable, time critical method of information transfer.

2.4 PC LANS

In the last decade local area networks have become a common tool within the industrial and commercial worlds. This trend has been fuelled as a result of the decreased cost of computer hardware and accompanied increase in its capability and has resulted in a number of changes in the way information is collected, processed and used. There is increasing use of small dispersed systems, which are attractive to users, since they are accessible and frequently more responsive and easier to use than large centralised, time-sharing systems. As the number of systems at a single site increase interconnection between systems provides the benefits of:

(1) sharing expensive resources

(2) exchanging data between systems

(3) facilitating management of the systems

These requirements necessitate communication between systems which may be met through local area networks (LAN).

Stallings [21] provides the following definition for LANs:
A local area network is a communications network that provides interconnection of a variety of data communicating devices within a small area.

Some of the key characteristics of LANs are that they are capable of high data rates (0.1 to 100 megabits per second) over short distances, usually no more than a few kilometres, with a low error bit rate.

A rigorously defined protocol is required to specify the format of messages and eliminate ambiguity to achieve reliable communication between nodes on the network. When a number of different kinds of devices from different manufacturers are to be interconnected it may result in complex protocol systems. A standard set of protocol layers is defined by the ISO to remedy this problem and facilitate design. The alternative LAN protocols that are available will be considered briefly. As an example, the choices that have been made for the MAP/TOP protocol, which places LANs in the context of manufacturing and process control, will be addressed . MAP was defined by a team from General Motors in an attempt to address the needs of manufacturing and process control using small networks of computers.

2.4.1 Local Area Network Protocols

In discussing LAN protocols we only need consider the first three OSI layers, the communication layers. The LAN protocols do not make full use of the ISO recommendations since it is sufficient to exchange a checked and reliable data packet stream between two nodes on the network. For LANs the physical and data link layers of the ISO definition may be sufficient, although some attributes of the network layer are required. Thus layering protocols for LANs have developed into a three layer system not directly equivalent to the three communication layers described earlier [22].

(1) *A physical layer identical to ISO layer 1.*

(2) *A medium access control (MAC) layer* to manage communications over the link.

(3) *A logical link control (LLC) layer to provide one or more service access points (SAP) which permit setting up a form of multiplexing to handle multiple-source multiple-destination data.*

Logical Link Control Layer.

The LLC layer is concerned with transmitting packet frames between two stations on a network that have no intermediate switching nodes. The layer performs error and flow control to provide acknowledgment and ensure error free transmission across the network. Two services that are similar to ISO layer three are a datagram service for unacknowledged traffic and a virtual circuit service to provide an acknowledged transport service.

Medium Access Control Layer.

For a given LLC protocol there are several different MAC options in which the LAN topology is taken into account. A bus/tree topology requires a suitable protocol to ensure suspension and retransmission of a data packet to avoid collision on the medium.

The most commonly used medium access control technique for bus/tree topologies is the carrier sense multiple access/collision detect system, which is one of a number of techniques referred to as a random access or contention technique. This process may be summarised by:

1) *If the medium is quiescent transmit the message.*
2) *If the medium is busy, pause until quiet then transmit immediately.*
3) *If there is a collision then wait a random time and retransmit.*
4) *If there is a collision then send a jamming signal to convey that the data has been corrupted.*
5) *Wait a random amount of time and start the process again.*

Different MAC protocols are required for physical or logical ring systems. This technique is based upon a token which circulates around the ring when all stations are idle. A station that wishes to transmit waits until it detects the token passing by. It then changes a status bit in the token from free to busy. Other stations must now wait until the token circulates back to the transmitting station at which stage the token is set free again. The next station in the ring will be able to seize the token and transmit.

The different protocols have implications for the applications that are supported across the LAN. The CSMA/CD protocol is non-deterministic and this may not suit a process control application (depends upon the timing requirements). The token ring/bus protocol is deterministic until the token is lost, and then the determinism is dependent upon the algorithm that will replace the token. A manufacturing automation protocol has been defined for manufacturing and process control applications by a team from General Motors. This protocol addresses the requirements of a LAN under these circumstances.

2.4.2 Manufacturing Applications Protocol

A single manufacturing organisation may have several systems, at the shop floor level, that are designed for specific tasks of computerised control and manufacturing. These systems are frequently incompatible although there is a definite need for integration in the interest of organisational and manufacturing efficiency. The automobile industry was one of the first to make widespread use, and suffer the inadequacies of, computer controlled production systems that are incompatible with one another. As a result, a team from General Motors produced a set of communications protocols, based upon the ISO seven layer model, for operation of a

local area network. These protocols are applicable to a wide range of equipment. The specific requirements for a LAN to operate at the shop floor are stringent and consist of [23]:

(1) *Support for a number of different types of devices.*
(2) *Provision for a range of services on the same network, e.g. speech, data, vision and control.*
(3) *Real-time communication.*
(4) *Communication between complex computer devices rather than between computer/terminal/peripheral.*
(5) *High reliability and absence of error to minimise faulty manufacture of machined or assembled units.*
(6) *Operation in a harsh environment, i.e. dirty, high level of electrical interference and supply voltage fluctuations.*

These requirements preclude the use of the carrier sense multiple access/collision detect (CSMA/CD) protocol since the risk of repeated collisions is too great for reliable operation, and efficiency deteriorates markedly under high loading conditions. The alternatives are the token bus or ring topology which can operate under more controlled conditions. The design team selected a broadband token bus system for the following reasons:

(1) To allow several devices to communicate simultaneously.
(2) To allow for a priority message operation.
(3) No minimum packet length.
(4) Reliable operation with high traffic loading.
(5) Calculable maximum waiting time for a given station.

MAP is designed to use a coaxial cable as the physical medium. The data rate is 10 Mbps which may be shared amongst a number of small sub-frequency bands to provide separate data channels.

Whilst MAP protocols need to be deterministic - one shop floor device must be able to communicate with another within a known maximum period of time - office routines can

accommodate small and variable delays in data transmission. For this reason, the technical and office protocol (TOP) has standardised on the cheaper CSMA/CD protocol.

MAP offers a high-performance protocol that is suitable for large organisations who can afford the associated high cost. The objective of this project however, was to produce a system that was cheap and readily available to facilitate real-time process control. Of the network protocols available in the Department of Electronic Engineering at the start of the project, NetBIOS best fit the requirements.

NetBIOS is a high level application programming interface for data exchange. Typically applications communicate across a local area network, although this is no restriction as two applications within the same machine or across different networks may communicate using fully acknowledged, connection-oriented or connectionless communication.

NetBIOS has become a de facto industrial standard with an interface appearing for TCP/IP environments, and operates under a number of operating systems, viz PC-DOS, OS/2 and UNIX. In addition there are NetBIOS emulators available for PC-based ETHERNET and ARCNET interfaces. NetBIOS was selected as the underlying communications protocol because it is widely accepted, readily available and it was felt that this would contribute to the environment's portability (a more exhaustive case is made for NetBIOS in chapter five).

The environment was implemented on an ARCNET LAN, a token bus system, running Novell Netware 2.15, since this was readily accessible within the Department of Electronic Engineering.

In summary this chapter has made a case for distributed systems in industrial applications. Ada, the programming language designed for embedded applications was noted for a number of strengths, in particular concurrent programming, exception handling, data abstraction and strong typing. However, Ada in its current form does not satisfactorily address distribution. Real-time systems are found to have complications that are not evident in non real-time systems. This requires a rigorous design approach to ensure a reliable system. The last section provided an introduction to local area networks. The requirements of a local area network

under manufacturing conditions was briefly mentioned and the manufacturing application protocol discussed. The following chapter will provide guidelines for writing an application designed to use the communications packages provided.

CHAPTER THREE

The Application Layer

3.0 INTRODUCTION

The design has taken the approach of distributing the functional units of an industrial process across a group of personal computers on a local area network (the advantages of this technique were discussed in chapter two). This chapter will suggest an approach for constructing an application package and in doing so specify the requirements for its interface. Later chapters will address the communications package itself and the degree to which this design has achieved its objectives.

3.1 THE APPLICATION LAYER

In designing a real time system using this environment, the application on a node forms a single part of the controller and must communicate with the other nodes of the system to achieve the overall objective of controlling the real-time process. As mentioned before, the objective of this design is to produce a communications package and a strategy for implementation of distributed process control systems for small industrial plants using Ada on PC LANs.

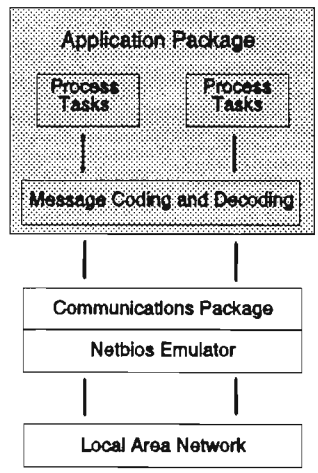

Figure 3.1 The Application Package

The communications package provides the facility to send and receive messages from other nodes on the system. Since the system is intended for general service, it must be the application's responsibility to produce a format for the message that is applicable in its situation

and will adequately convey information to other applications. Consequently the application may be partitioned in two ways: there is the control application section which controls devices, makes calculations and collates information and there are the coding and encoding routines that the application uses to manipulate messages for sending and receiving (figure 3.1). The specifics of the control application section cannot be dealt with here since that is dependent upon the process to be controlled. Rather, the *caveats* that are applicable to typical systems of this nature will be mentioned.

3.1.1 The Control Application Section

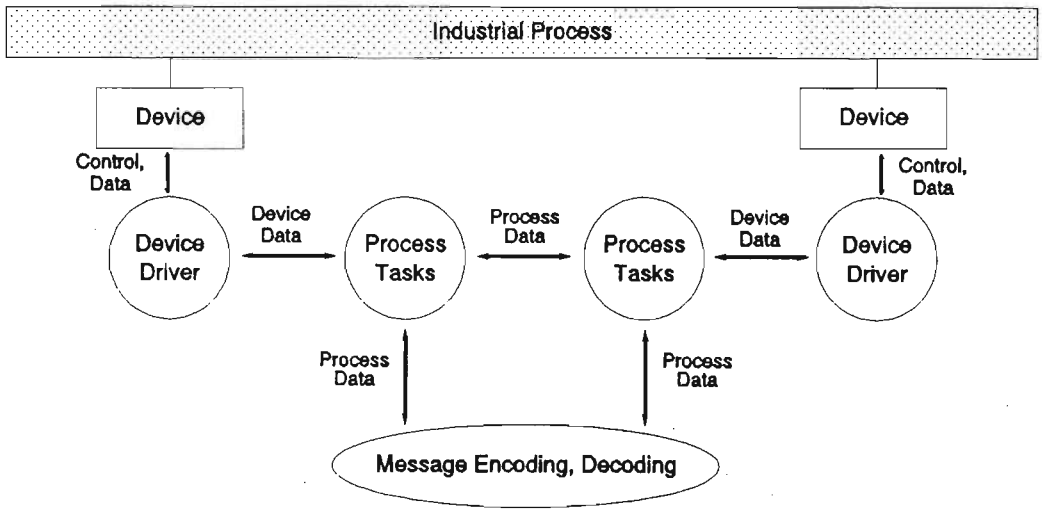

Figure 3.2 The Interelationship of Units within the Application Layer of a Single Node

A successful real-time system will make optimal use of the capabilities of the hardware devices involved and will satisfy the desired performance requirements. The application that exists at a single node may be viewed as a microcosm of the entire real-time system and must be decomposed into a set of software modules that may be implemented in Ada packages, tasks and subprograms (figure 3.2).

Some of the additional difficulties in designing a real-time application are [24]:

1) *Processing of data arriving at irregular intervals, fluctuating rates and priorities.*
2) *Control of fault conditions with recovery if feasible.*
3) *Handling of queues and buffers for storage of messages and data.*
4) *Modelling of concurrent processes into multiple threads of execution.*
5) *Treatment of communication and synchronisation between tasks on remote nodes.*
6) *Management of strict time requirements and performance specifications.*

When implementing a controller the process control engineer must design an application package that allows the system to run in an automated fashion with extremely high reliability. This is important both from a performance and cost point of view as a plant with frequent shut-downs is not very cost effective. Furthermore, the application must be designed with a high degree of correspondence and simplicity [18]. Correspondence in this context refers to how closely the solution maps onto the problem space and aids the determination of the correctness of the solution. A simple design makes the solution easy to understand and maintain, and enhances reliability.

Device Drivers and Interrupt Handlers

An application will interface with hardware devices to receive data from sensors and output control information to device controllers. The design considerations for input and output device drivers differ and must be approached separately. Attention must be given to the following points of interest in input device drivers [19]:

1) *The information may arrive in a scheduled or irregular manner, in continuous or burst transmission.*
2) *Is the application to poll the input device or is it to be notified through interrupts?*

3) *Must the application move the incoming data to a buffer for intermediate storage or is it capable of completing the required processing in time? (If buffers are used, a protocol must exist to manage the case where the buffer is full).*
4) *Error detection and handling facilities must be available.*
5) *Failure detection of an input device.*

The design considerations for an output device are less complicated and include:

1) *Selection of an appropriate protocol (e.g. handshaking requirements).*
2) *Error detection and handling.*
3) *Failure detection of a device.*

Ada does not have a real-time executive nor a supervisor program, rather the real-time facilities of task scheduling and dispatching are reflected in language constructs and supported by the run-time system. Interrupt handling is offered by Ada constructs (task interrupt accept) and normally written into the device driver as opposed to a separate interrupt handler. Meridian Ada does not offer the facility for an input parameter with its interrupt entries that is permitted under the Language Reference Manual definition [1]. It is usual to encapsulate a device driver within a task as part of a set of concurrent processes.

The application should make use of package constructs which are the primary means of creating modules in Ada. They are passive software components that are elaborated only when an Ada program is being prepared for execution and their primary function is to encapsulate active software units such as tasks and subprograms.

Tasks are provided by Ada to allow multiple threads of execution within a single program. Even within one node, the application programmer is expected to make use of the tasking construct to model concurrent processes, such as monitoring a number of devices, collating data and performing calculations. However, true concurrency is not possible on a single CPU and the Ada run-time system must approximate this using some form of task scheduling. The performance of any real-time system is inextricably tied to that of the run-time system and its

scheduler, under which the real-time system operates. The run-time scheduler for an Ada compiler is not defined by the Ada Language Reference Manual and depends upon the implementation of the compiler. Ada9X is expected to introduce further guidelines on the specification of the run-time scheduler, since the design and portability of a system can be materially affected by this choice. The Meridian Ada 4.1 implementation offers two distinct scheduling alternatives [25]:

1) *Prioritised pre-emptive scheduling with round-robin time-slicing for equal priorities.*
2) *Non pre-emptive scheduling (run until blocked) with prioritisation.*

The issue of scheduling and pre-emption (and task priorities) will be considered more fully in a later chapter. Task priorities and priority inheritance, and consequently pre-emption, are under much discussion within the Ada community and will most likely undergo changes in the new revision of Ada, Ada9X. The current version of the communications package has been written with a run-time system that uses the round-robin, non pre-emptive scheduler. Synchronisation is managed through critical regions to ensure some measure of compatibility. Non pre-emptive scheduling implies that the processes, interrupt, clock or base level, run to completion without interruption. Upon completion, each process transfers control to the dispatching mechanism. Significant advantages, as documented by Allworth [18], are obtained when using this technique. Non pre-emptive scheduling prevents corruption of shared data owing to poor synchronisation, shared subroutines may be implemented without producing re-entrant code or implementing lock and unlock mechanisms, and it is simple and efficient. The code must still be designed to cope with an alternative scheduler (e.g. pre-emptive, with or without time-slicing) so access to shared variables must be carefully controlled through semaphores or guard tasks.

Meridian provides an implementation of pre-emptive scheduling that has a slightly negative effect on program performance in general [25]. The main drawback with this approach is that it can have a poor response to changes in the environment, although this may be overcome with careful programming (context switching may be forced at synchronisation points and with a

delay one_clock_tick statement). These issues will be discussed in more depth when there is a deeper understanding of the construction of the communications package and its utilisation.

3.1.2 The Message Encoding and Decoding Routines

Since the communications package passes messages, and the application frequently requires the results of calculations and synchronisation information to be communicated to a remote node, some method is required for transforming the numerical data into a format that may be sent as a data message. However, the primary inefficiency in a distributed system is the amount of message communication that takes place between processors. The more data that is passed between processors, the more overhead, reducing the overall efficiency of the system. The functional decomposition and the message format must be such that it reduces the data traffic to a minimum.

In this design, as in the case of some disk systems, to increase efficiency, fixed length messages are used for records. The maximum length message is specified for the entire environment (to simplify message management) and is fixed by the application programmer. Since all nodes may communicate with one another if required, information can be directly transferred to the target destination.

It is envisaged that the types of data passed between nodes through messages consist of the base types that exist for an Ada implementation (i.e. types boolean, integer, float and character). Any composite types may be reconstructed at the destination node. The great weakness in such a message passing system, is that there may be an incompatible type allocation at the decoding or encoding stage if care is not exercised. This problem may be alleviated to some extent through a small amount of preprocessing to check whether the types are comparable. Alternatively, variable types can be checked after opening a channel by first sending an initialisation message with type information. Furthermore, it is desirable to keep the encoding and decoding procedure as simple as possible to reduce the amount of run-time overhead this incurs. The actual conversion of the string to the base type can be achieved through the use of the generic package *unchecked_conversion* which maps the actual bit pattern from one type onto another. Obviously the number of bits in the representation of the two types must match. This

conversion does not guarantee that the conversion will produce a compatible type and the application ought to provide an exception handler to deal with error conditions. This approach can lend itself to misuse so it should be used with great care.

In summary the message encoding/decoding routines must perform simple operations as quickly as possible to reduce the overhead of the message passing system. This task is greatly simplified if a careful study is made of the communication requirements before the routines are constructed. Some guidelines are:

1) *The system must be decomposed to minimise the inter-processor communication.*
2) *The message size should be chosen to fit the data that needs to be transferred while optimally utilising the packet sizes of the lower layer communication protocols.*
3) *Use a simple decoding/encoding operation to minimise the calculation overhead.*
4) *The message buffer should be as small as possible to conserve memory space, provided it does not severely impact upon the performance of the application.*

3.2 THE COMMUNICATION INTERFACE DEFINITION AND SPECIFICATION REQUIREMENTS

A process control application operating over a distributed system requires a communication system to pass information from one processing node to another. This section shall provide the communication interface definition and specification requirements.

The communication package will be a generic library:
The use of a generic package with input parameters, will allow the user to tailor the communications system for each individual node. This may be achieved with minimum effort by altering the generic parameters during instantiation of the library for each respective processing node (figure 3.3).

```
Generic
    maximum_message_length : integer;
    maximum_buffer_size : integer;
Package Communications is

    .....
    ...    Communication
    ...  Function Definitions
    .....
End Communications;
```

Figure 3.3 The Generic Communications Library

The communication package shall use an available, underlying communications protocol to achieve the transfer of information between two processing nodes:

This project will use tools that are currently available to improve the portability and reduce the cost of implementation of the communication system. NetBIOS has been chosen as the underlying protocol (this will be motivated extensively in the following chapter). NetBIOS session support provides a connection-oriented service that will ensure successful transfer of information. In process control systems the overhead of a connection-oriented interface is acceptable where it is necessary to ensure guaranteed delivery through positive acknowledgement of packet delivery. NetBIOS is widely accepted as a de facto industrial standard.

The communications package will possess functions that are used for initialisation and management of the network, data transfer and facilities that aid in the debugging of the communications system:

The communication interface must offer facilities to initialise, construct and close communication channels as is required by a connection-oriented interface. Functions are required to transfer data to fulfil the purpose of a communications system. Furthermore, the use of NetBIOS as the underlying protocol will necessitate some functions that are NetBIOS specific. Debugging facilities will simplify the use of the communications package in a process control system.

3.2.1 Network Management Functions

The communications interface will provide a function call to allow the application to determine whether NetBIOS exists at the node:

Since the communication system is based upon the NetBIOS protocol it is essential to establish that NetBIOS exists at that node in the form of a NetBIOS ROM or emulator, before an application executes. A function, *check_for_netbios*, is required to return a boolean as a result (figure 3.4).

```
-- Does NetBIOS exist at this node
if check_for_netbios then
    -- Perform the setup, return a failure code to
    -- indicate the status, and specify the retry
    -- count, i.e. the number of times the system must
    -- attempt to establish a channel before failing
    comms.setup( failure_code => code,
                 retry_count => retry_attempts);
else
    tty.put_line("NetBIOS does not exist ");
end if;
```

Figure 3.4 Communication Initialisation

The communications package will provide an initialisation call to allow the application to specify when the communication network should be instantiated. This call will include a simple mechanism for the creation of the communication network:

An initialisation call permits the application to specify when the communications system is to be established. In connection-oriented communication the application must initiate itself as a valid partner on the network (i.e. a name that may be addressed by other applications) before any communication is attempted. Furthermore, a single call for the construction of all the communication links to other nodes removes the necessity for the application to explicitly create each individual channel. The *system_setup* call is the routine that fulfils these requirements (figure 3.4).

The communication package will provide further calls that allow the application to instantiate a communications channel, close a channel, obtain a list of the valid partners on the network, obtain the applications own host name and reset the communication system:

A connection-oriented communication system requires facilities to identify and name a partner. The communication package must create a virtual circuit between two partners on the network to allow information to be transferred. Furthermore, the communication package must supply the status of the other partners on the network to allow the application to make an informed decision (figure 3.5).

```
-- Initiate a communication channel
procedure Initiate(name : In string;
                   failure_code : out integer);

-- Close a communication channel
procedure hang_up(name : In string;
                  code : out Integer);

-- Reset the communications network
procedure reset(failure_code : out integer);

-- Fetch the applications node name
function my_name return string;

-- Return an array of the valid partner names
procedure obtain_connection_array(name : out name_array
                                  lgth : out integer;
                                  code : out Integer);
```

Figure 3.5 Network Management Functions

The communication package will provide a facility to redirect the output of a node from one target to another:

To facilitate the insertion or replacement of a processing node, the communication package is required to supply a mechanism for an application to redirect the output of a target node to another destination. The redirection facility will allow an application to redirect the output of two adjacent nodes to itself, and so insert itself within the network. The process will be achieved while the control system is on-line, with the minimum of disruption to the processes already executing at the existing nodes. The *redirect* command will allow an application to replace a node already in existence or to introduce a new function (figure 3.6).

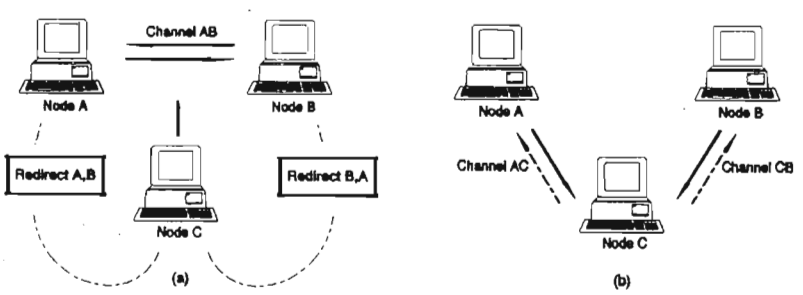

Figure 3.6 The Redirection Operation

3.2.2 Data Transfer Functions

The communication package will supply the facility to send a message to a named destination node. The application will select a number of options that reflect the importance of the message. Where appropriate additional calls will be provided to return a failure code that indicates the status of the transaction:

To perform distributed process control the application needs to transfer information from one node to another across the network. An asynchronous send is necessary since it permits the application to continue processing as soon as the send command has been pended (i.e. enhances concurrency). A function *get_send_failure_code* is required so that the application can fetch the failure code for the asynchronous case. The synchronous send is necessary where no further action can be taken until the success of the immediate transaction is ascertained (figure 3.7).

```
-- To send information to a named node.
-- Discard_msg and Ack allow the
-- the  priorities of the msg to be established
send(node_name : in string;
          message : in string;
     discard_msg : in integer;
              ack : in integer;
     failure_code : out integer);


-- To allow the application to retrieve the
-- failure_code for the asynchronous send
function fetch_send_failure_code  return integer;

-- Fetches a message from the message buffer
get_latest_message( message : out string;
                    buffer_position : out integer;
                               name :  out string;
                       failure_code : out integer);
```

Figure 3.7 The Send and Receive Functions

The communication package will supply the application with a facility to retrieve messages that have been sent to it from a remote node:

The application must receive and decode to perform useful control. The communication package will interrogate a buffer to obtain the latest message and failure code for the associated partner (figure 3.7).

The communication package will supply a facility to despatch emergency messages to a destination node. The emergency message must signal the application and allow user defined routines to deal with the situation:

In extreme situations an application may need to despatch an emergency message with the knowledge that when it is received the current process will be halted and the new problem evaluated. The send routine will allow the user to specify an interrupt entry for an Ada task at the target (figure 3.8).

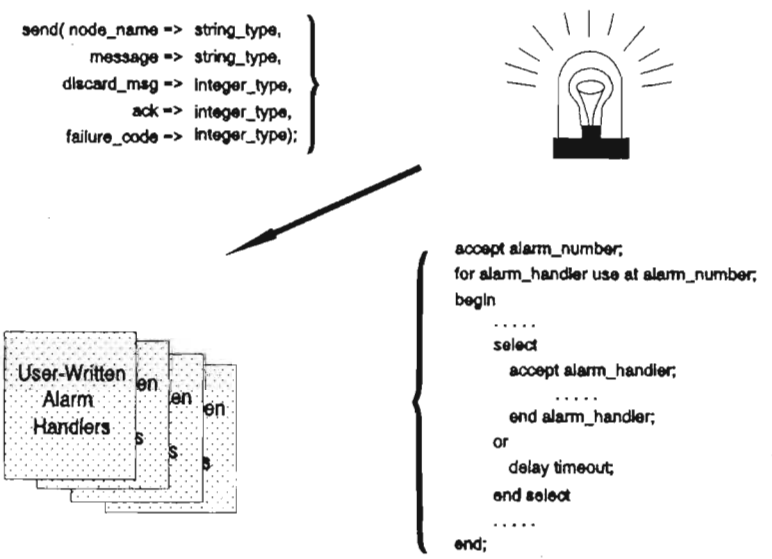

Figure 3.8 The Alarm Handler Mechanism

The communication package will not supply a send to multiple destinations:

A multiple send routine has not been provided since this may be implemented and controlled more effectively by the application through explicitly despatching the message to each respective target.

3.2.3 The Debugging Functions

The communication package will include debugging calls to assist the application programmer in implementing the communications system:

Additional calls to display the state of the communications system buffers facilitate the debugging of the communications system. The application will require a simple mechanism for

obtaining a complete list of all the messages and their status in the message buffer. Furthermore, a convenient method of viewing the status of all the valid communication channels is necessary. The communications package will provide calls for a visual display of the latest message received and the full contents of the message buffer. This will list the message, the place it occupies within the buffer, and whether it has been read. Another call will list all connections that the node has open with other nodes on the system. These calls are supplied to facilitate the debugging of the communications environment (figure 3.9).

```
-- This call will display all the messages stored
-- in the message buffer

Procedure message_display;


-- This call will display all the names of partners with
-- valid communication channels on the network

Procedure con_display;
```

Figure 3.9 The Debugging Calls

This chapter has examined some of the issues that need to be considered when constructing an application package and has motivated the communication interface that is supplied with the communication package. A complete specification and user manual for the interface is contained within Appendix E. The following chapter will discuss the underlying communications protocol chosen and will be followed in the next chapter by a discussion of the actual structure of the communications package, motivating the design decisions that have been made. Two case studies will be considered in the chapters that follow.

CHAPTER FOUR

The Communication Protocol

4.0 INTRODUCTION

When the process control software environment was designed a decision had to be made whether to design a communications protocol and write the applicable network drivers to implement it or to make use of an existing protocol. Owing to a lack of manpower and time, and the desire to maintain as high a level of portability as possible, it was decided to select an existing network protocol in preference to writing a new one. The appropriate protocol selected had to be

Readily Available.
Compatible with the available hardware and software (Novell Netware 2.15).
Widely used - to facilitate portability between LANs.
Would operate under real-time constraints.
Well documented.

The protocols that were readily available on the University of Natal's, Department of Electronic Engineering ARCNET local area network and that fulfilled the above criteria at the time the work on this thesis started, were IPX/SPX [26,27,28] or NetBIOS [29,30].

4.1 THE NETWARE INTERNETWORK PACKET EXCHANGE PROTOCOL (IPX)

This is an implementation by Novell of the Internetwork Datagram Packet protocol designed by Xerox and described in their *Internet Transport Protocols* document [28]. The purpose of IPX is to allow applications running on Netware workstations to take advantage of the Netware communications drivers to communicate directly with other workstations, servers or devices on the internetwork.

It allows applications to send or receive individual packets that are structured according to the Xerox Network Standard. The network drivers make a best effort to deliver packets but offer no guarantee of delivery, in a connectionless environment; reliable delivery, sequenced protocol, data streams must be built upon the IPX services.

To this end Novell have provided enhancements in the form of the Netware Sequenced Packet Exchange Protocol (SPX) which is implemented upon IPX. This guarantees delivery of the packets by using the IPX primitives to send and receive acknowledgement messages in a connection orientated interface. Packets that are not acknowledged are retransmitted by SPX and if, after a number of retransmissions there is no positive acknowledgement the connection is assumed to have failed. Appropriate time out and re-transmission values are selected by SPX to match the physical characteristics of the intervening networks.

SPX uses a data structure containing relevant transmission information, called the Event Control Block (ECB), with which to issue commands. For a send command, fields in the ECB point to a buffer that contains the data to be sent and the destination. It is possible for an application to specify that an event service routine ought to be executed upon completion of a command (this corresponds to an interrupt handler). In summary IPX/SPX provides primitives that enable the application to maintain a connection oriented communication channel and to send or receive packets of data in a reliable fashion; it is also possible to have event driven occurrences upon the conclusion of a command. The reader is referred to [26] for further detail.

4.2 NETWORK BASIC INPUT/OUTPUT SYSTEM (NETBIOS)

NetBIOS is a high level application programming interface for data exchange. It provides services both to identify sources and targets across a LAN and to transfer information. Typically applications invoke these services to communicate across a local area network although this is no restriction as two applications within the same machine or across different networks may communicate.

In the original IBM design, NetBIOS was intended to fit within the transport and session layers of Open Systems Interconnection (OSI) reference model of the International Standards Organisation (ISO) . Each layer directly coordinates communication with the adjacent layer and indirectly with its peer in the other machine. The applications that are programmed to the NetBIOS interface standard are insulated from the lower protocol layers and their interaction with the precise communication protocols used and the physical network.

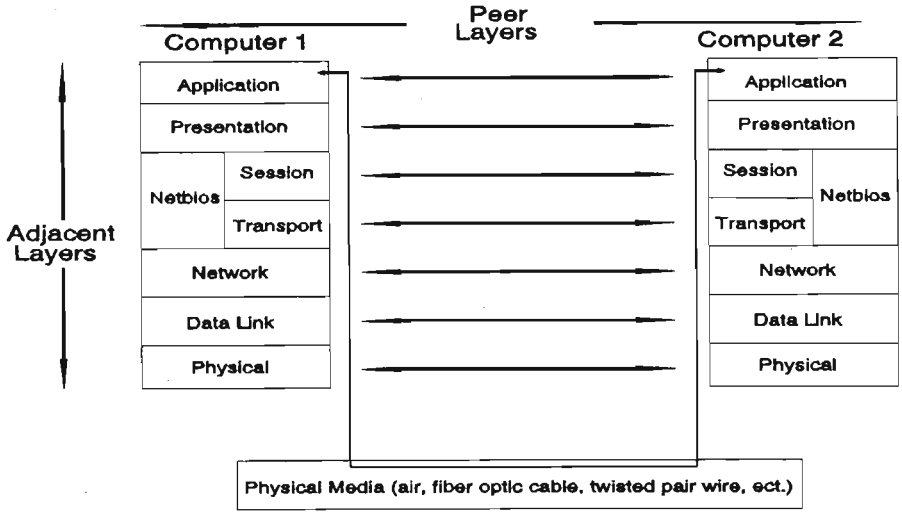

Figure 4.1 ISO/OSI Reference Model
Taken From Schwaderer [30]

The original IBM version of NetBIOS supports five layers, the physical, link, network, transport layer and session layer. The layers are intended to provide a reliable virtual connection service, name support facility and a connectionless, low overhead datagram service. The BIOS provides the user interface to application programs.

Two types of data transfer are supported. Reliable transfer of information is supplied by the session layer where, if data is lost or corrupted, the BIOS returns an error code. Datagram support accesses the link layer directly and offers only a "best effort" attempt at transferring the data. The most common use for this type of approach is in the broadcast of messages. All communication in NetBIOS identifies the destination and source through names, of sixteen bytes in length, which are added before communication begins. NetBIOS possesses a NetBIOS Control Block in the same fashion as IPX possesses the event control block and the application may specify an interrupt handler to be executed, if desired, upon completion of a command as in IPX.

4.3 A CONNECTION-ORIENTATED OR CONNECTIONLESS SYSTEM

Both the Novell protocol and that of NetBIOS offer both a connection orientated message passing system that acknowledges successful delivery or returns an appropriate error code, and a connectionless system.

The connectionless method is one of the oldest and simplest technologies. It treats each packet as an individual entity which contains both a source and destination address and the data to be transferred; this is referred to as a datagram. The connection overhead in the implementation of workstation network shells, internet routers, bridges and servers is completely eliminated. The system may duplicate and transmit the message to many different destinations by simply altering the destination address. The simultaneous availability of both the sender and receiver is not required if the network components can store the packets. The simplicity and greater flexibility of the connectionless method offers a high performance solution for packet networks.

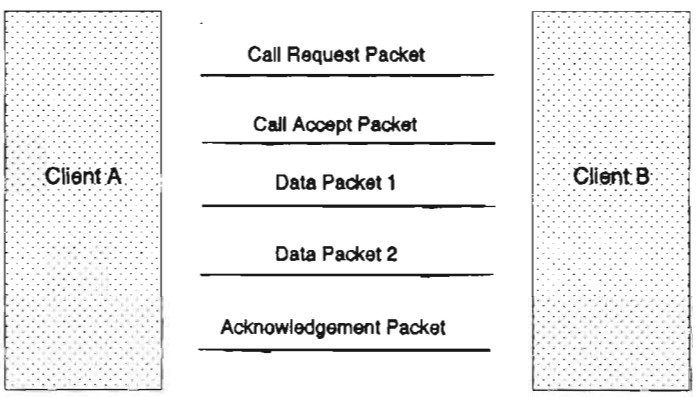

Figure 4.2 Connection Orientated Communication
Taken From Lee [28].

Connection-orientated communication is generally more reliable, but imposes a certain amount of overhead that may not be desirable (figure 4.2). In fact, connectionless communication may be more efficient if the sender-receiver handles error conditions properly. The call request, call accept and call acknowledgement packets illustrate the amount of connection overhead in connection-orientated communications. The virtual connection interface is built on top of the datagram interface and offers guaranteed delivery by providing a positive acknowledgement of packet delivery.

NetBIOS and IPX/SPX offer very similar services that differ, in most cases, in name only. NetBIOS has become a de facto industrial standard with an interface appearing for TCP/IP environments, and operates under a number of operating systems, viz PC-DOS, OS/2 and UNIX. In addition there are NetBIOS emulators available for PC-based ETHERNET and ARCNET interfaces. NetBIOS was selected as the underlying communications protocol because

it is widely accepted and it was felt that this would contribute to the environment's portability. The environment was implemented on an ARCNET LAN running Novell Netware 2.15.

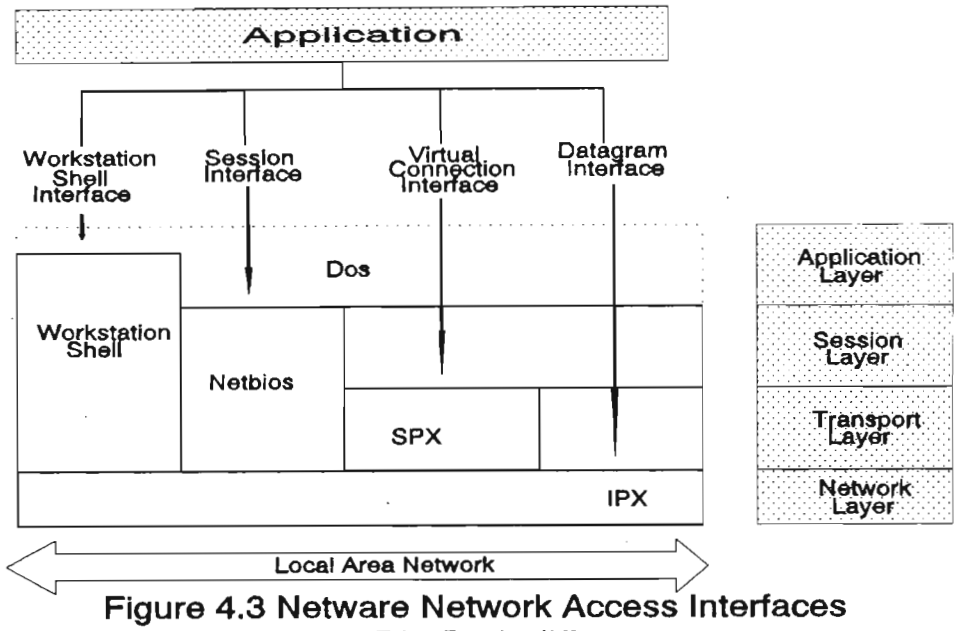

Figure 4.3 Netware Network Access Interfaces

Taken From Lee [28]

Session communication is most suited to an environment where, as in process control, it is necessary to guarantee that communication is maintained so informed decisions may be made on the status of the industrial process at hand. Since these assurances would need to be obtained in any event the utilisation of the session layer is appropriate. The NetBIOS interface, the session layer, is also built upon the datagram interface to allow network applications designed to use NetBIOS to run without modification. Sessions provide reliable point to point, full-duplex virtual circuits.

4.4 NETBIOS SUPPORT ENVIRONMENT

The NetBIOS environment provides four categories of application services, name support, datagram support, session support and general command support.

4.4.1. Name Support

An individual NetBIOS adapter is distinguished from other adapters across the network through its name. This name allows an adapter to direct a message to a particular destination and to indicate a source from where the message originated. NetBIOS names are case sensitive and

sixteen characters in length. Before an adapter may use a name it must acquire the rights to register it. The adapter issues either an "add name" command or an "add group name" command. In the first case the name must be unique and is registered after it is broadcast throughout the network and the adapter has obtained an assurance that this name is not in use by any other adapter. In the latter case the adapter must obtain an assurance that the name does not exist on the network or exists only as a group name.

If the registration fails then this is reported for analysis to the workstation along with an appropriate error code, otherwise the name is registered and assigned a single byte NetBIOS name number, in the NetBIOS name table. This table is scrutinised when a petition for registration is received from another adapter. Each adapter has its own name table, which is temporary and requires no central name administration.

4.4.2. NetBIOS Datagram and Session Support

Once a name has been added the application may begin to communicate with other adapters on the network, through datagrams or sessions.

Datagrams are short messages of up to 256 bytes and the sender has no guarantee of successful delivery. The intended destination may not exist, may be powered off, not be prepared to receive a datagram or may receive the message successfully with no indication available for the source adapter. There are two types of datagram communication: broadcast datagrams and plain datagrams. In both cases the adapter references the name number of the name added to the local name table. The destination must have pended a datagram receive command for the message to be successfully received.

NetBIOS also offers session communication. The adapter creates a reliable, full duplex connection. The communicating applications may reside within a single adapter or across several adapters. The advantage of session communication is that a message receipt status is issued to the transmitting application for each message; the message size may also be up to 64 kilobytes. The life-cycle of a session begins after the adapter is reset and a name has been added to the NetBIOS name table (figure 4.4).

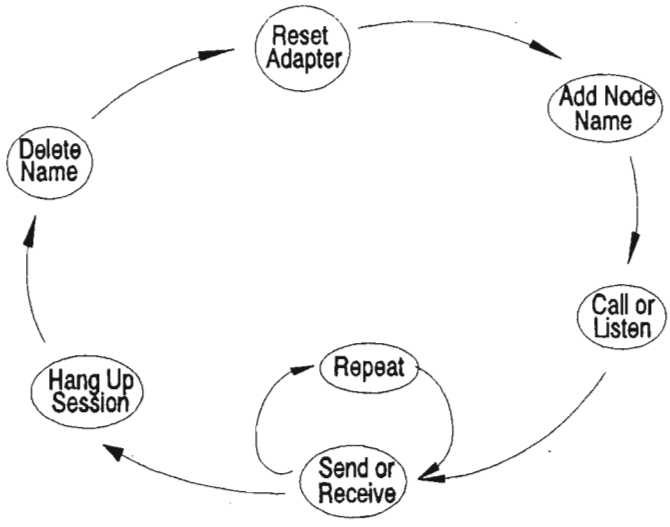

Figure 4.4 Life Cycle of A Netbios Channel

Sessions are created when one application issues a NetBIOS listen command that the other application "calls". NetBIOS does not permit the sequence of the listen and then the call to be reversed. A session starts when both commands complete successfully and return a local session number to the applications. The maximum number of sessions that may exist simultaneously is 255 and the receive and send commands are required to quote the session number when they execute. The application may specify the length of time NetBIOS will wait before it determines that the connection has been broken and is no longer valid. When the applications have completed their data transfer they may indicate to NetBIOS that the session is no longer required and may be closed. A NetBIOS "hang up" command is issued to close the session. If all sessions are closed then the NetBIOS name may be deleted and the adapter reset.

4.4.3 The NetBIOS Control Block

NetBIOS commands make use of a block of data called the NetBIOS Control Block to issue commands (figure 4.5). The command field is a one byte field containing the NetBIOS command code. A listing of all the command codes may be found in Appendix A.

NetBIOS permits the majority of commands to be issued in three ways. If the high order bit of the command code is set then NetBIOS returns to the application to allow it to continuing processing, immediately after pending the command. This is referred to as the no-wait command option; when the command completes the final completion code is stored in the

NetBIOS command complete field. In this way a number (dependent upon the application - with a maximum of 255) commands may be pending simultaneously. Obviously the block of memory that contains the NetBIOS control block must be preserved until the command completes.

If the high order bit is not set the application will only resume processing when the command times out. The time out length is set when the call and listen commands are pended; if the no time out is specified and the command cannot complete then the PC will hang causing the application to fail. This is known as the "wait until completion" option. Clearly only one "wait until" command can be pended at one time. Not all commands may operate the "no-wait" option since some of the commands (e.g. reset) are guaranteed to complete under any conditions. If NetBIOS accepts the command it queues it for subsequent action.

There is one final variation on the "no-wait" option known as the "no-wait with interrupt". When the command completes NetBIOS checks to see whether the post-routine contains all zeros; if it does not execution begins at the address specified in the post routine field as it would in an interrupt handler - requiring the post routine to complete with an assembler "iret" instruction. The remainder of the control block fields are discussed in Appendix D.

4.5 NETBIOS IN THE SOFTWARE ENVIRONMENT

NetBIOS session support is used to establish peer to peer communication channels between those process applications that exchange data during normal execution. Two sessions are created per communications channel for each direction of transfer. A library of NetBIOS functions implemented is Ada has been written and is included in Appendix C. The library package returns error codes based upon those offered by NetBIOS and included in Appendix B. Appendix A has a list of the NetBIOS command codes.

In this chapter, a case has been presented for adopting NetBIOS as the communications protocol that the communication system uses for inter-processor communication. A failure code is returned by the communication system that uses the NetBIOS error codes to create an extended error table listed in Appendix B. The next chapter will discuss the structure and design of the

communications package, and how it interface with the NetBIOS communications protocol. The structure of the library, and its relationship with the communications package, and application will be considered in the next chapters.

CHAPTER FIVE

The Communication Package

5.0 INTRODUCTION

The nodes in a distributed system execute concurrently and process functions existing on these nodes must be able to pass data to one another through messages, and synchronise their activities for meaningful industrial process control to be accomplished. The logical structure of the process controller may be viewed as a network of software components that are connected via the communication system. The communications layer offers two basic functions: peer to peer communication to allow transfer of data and synchronisation information, and network configuration control. Peer to peer communication is established only for those processes that must exchange data during their normal operation. The communication facility supplied uses NetBIOS sessions and is reliable, with acknowledgement of successful transactions, though a connection-oriented interface is somewhat expensive (motivated in the previous chapter). The application is offered an interface to the communications library, through which the network configuration is manipulated, and data is sent and received in the form of messages.

5.1 REQUIREMENTS

The communication system must fulfil the requirements posed by a distributed process control system operating in an industrial environment as set out in the previous chapters. The implications of these requirements are such that a communications system must support the implementation of a network of PCs and allow for processors to be added or removed and new communication links to be established. Clearly, the nodes are loosely coupled in the sense that they can continue with local processing during such reconfiguration activities. There must also be the facility for synchronisation signals and data to be received and transmitted reliably.

5.2 STRUCTURE AND OPERATION

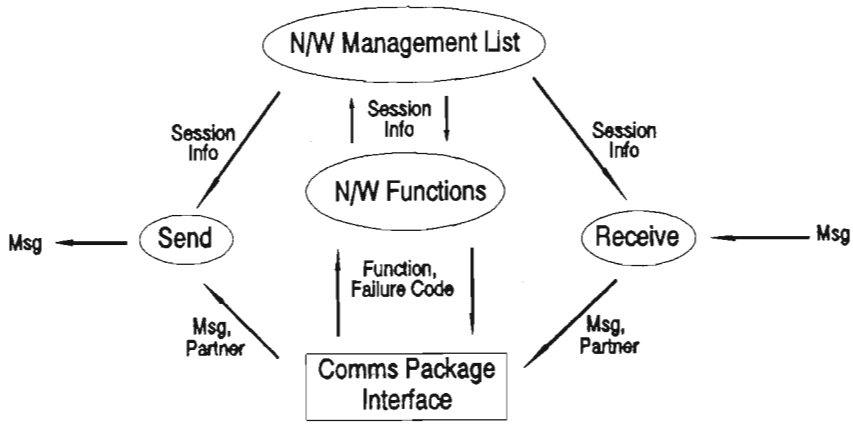

Figure 5.1 The Communication Package Data Flow

The structure and operation of the communication system may be described in terms of the functions that are performed by it. Those functions that ought to be accomplished concurrently, and repeatedly, are encapsulated in Ada tasking primitives which maintain an environment that may be accessed via service calls from the application, or through NetBIOS interrupts from within the lower communications protocol. The tasks provide the working environment of the communications package.

Any communications system that makes use of a connection-oriented interface, requires facilities to perform initialisation, to make and break the inter-nodal connections and to send and receive information. The structure of a communications system may be effectively visualised by dividing it into four segments; viz a receive, a send, a network function and network list segment (figure 5.1). The management of the communication network is controlled through the use of function calls, provided through the communications interface which allows the application to instantiate and remove connections to nodes on the network. They require information (such as session numbers and partner names) in order to process requests.

The communications package is structured such that all functions access a Network Management List to obtain information on communication channels, or to alter the status of these channels. Access to this list is controlled by a management task which orders the sharing

```
Task manage_network_list is
    entry startup(length_list : in integer;
                  host : in node_list_pointer;
                  lp : in node_list_pointer);

    entry lsn_write(name : in node_name_type;
                  lsn : in byte;
                  flag : out boolean);

    entry get_connection_array(name : out definitions.name_array;
                  ingth : out integer;
                  failure_code : out integer);

    entry get_lsn(name : in node_name_type;
                  np : out node_list_pointer;
                  flag : out boolean);

    entry get_name(lsn : in byte;
                  name : out node_name_type;
                  flag : out boolean);
    entry delete_connection(lsn : in byte;
                  flag : out boolean);

    entry display;
end manage_network_list;
```

Figure 5.2 The Manage Network List Task Specification

and alteration of the information. The specification of that task is included in figure 5.2 (the *display* entry dumps all information to the display -- it is a debugging facility).

The communications package also has an initialisation routine which is executed before any other communication package transactions. It allows an application to specify when the communication system should be instantiated and performs a sequence of statements that initialises the NetBIOS interface and starts the communication package tasks in an orderly manner. Once the initialisation has completed, the send, receive and network functions of the communications system can be viewed as essentially separate entities, within a single node, that are linked by the application, their common need for information from the network management task and when the system is terminated (figure 5.3). They are encapsulated in tasks to give apparent concurrency. Before execution of the application the programmer is expected to create a configuration file, *netcon.cfg*, with the netcon.exe utility. This file contains the network node name and the name of any prospective partners. If partner names are included (not essential) the communications initialisation routine automatically begins the instantiation of the communication channels. It was felt that this option relieved the programmer of some tedium, yet, if it was desired, the application could initiate the channels explicitly through the

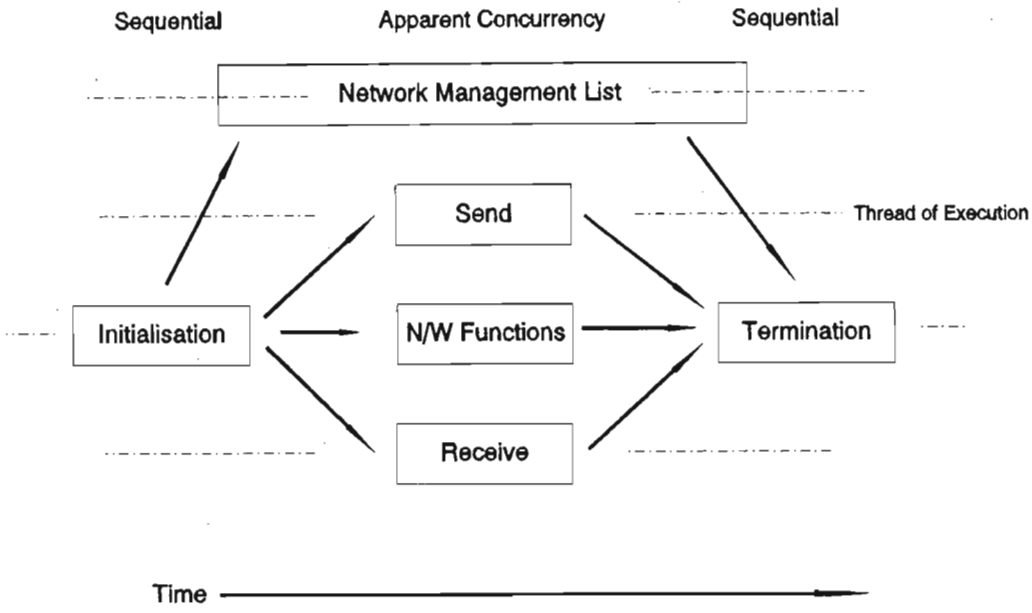

Figure 5.3 The Communication Package over Time

initiate (a network function) call. The initialisation of the system has completed once the routine has performed the following tasks successfully:

(1) Obtained the network node name from disk.

(2) Obtained and created a list of prospective partners.

(3) Reset the NetBIOS adapter.

(4) Added the network node name to the NetBIOS name table.

(5) Initialised the *manage network list task* by passing it the list length and a pointer to the list.

(6) Started the *listen task* to allow the node to create communication channels with a node that calls it.

(7) Started the *receive message task* to allow the system to receive any messages that are sent to it.

(8) Initialised the *send task* to manage any messages despatched by the local application.

(9) Began the automatic creation of communication channels with the partners specified on a disk file.

(10) If a fatal error was detected halted the system and aborted the tasks.

5.2.1 The Network Functions.

The design of the communications environment had to incorporate a number of functions that allowed the communications package and the application, through the interface, to manipulate the communication channels on the network. Some of the network functions are structured as simple procedures. For example, the *reset* function initialises the network adapter by constructing the NetBIOS control block (discussed more fully in Appendix D) and issuing interrupt 5Ch. The NetBIOS *reset* and *add name* commands are issued with a "wait" alternative and the procedures return with the failure code presented in the NetBIOS control block. The more interesting network functions and their designs are considered below.

One of the first duties of the initialisation routine, as listed above, was to start the listen routine. Connection-oriented communication may be viewed as a virtual circuit between two nodes. To achieve this the NetBIOS interface employs a listen and call process. NetBIOS requires the listen command to execute before the corresponding call for a connection to successfully occur. The design had to ensure that a new connection could be instantiated with minimum delay and disruption to the application services.

The listen routine was encapsulated in an Ada task to provide it with its own thread of execution and give the appearance of concurrency. The NetBIOS listen command was pended with a "no-wait interrupt" option so a user-defined routine, to update the network management list, could execute when the command completed. A problem existed since the NetBIOS post routine fields in the control block require an offset and segment address of the interrupt handler and Meridian Ada does not permit obtaining the address of a procedure with the address attribute. An inelegant, although effective solution was found through using an Ada task interrupt entry. The address of a string of bytes, which contained the machine code for a software interrupt to the Ada interrupt entry was passed to NetBIOS as the post routine address, which in any event expects an *iret* instruction to complete that handler. Figure 5.4 documents the process.

The listen routine is structured such that it executes independently of most of the other functions in the package. It communicates only with the NetBIOS adapter and with the network

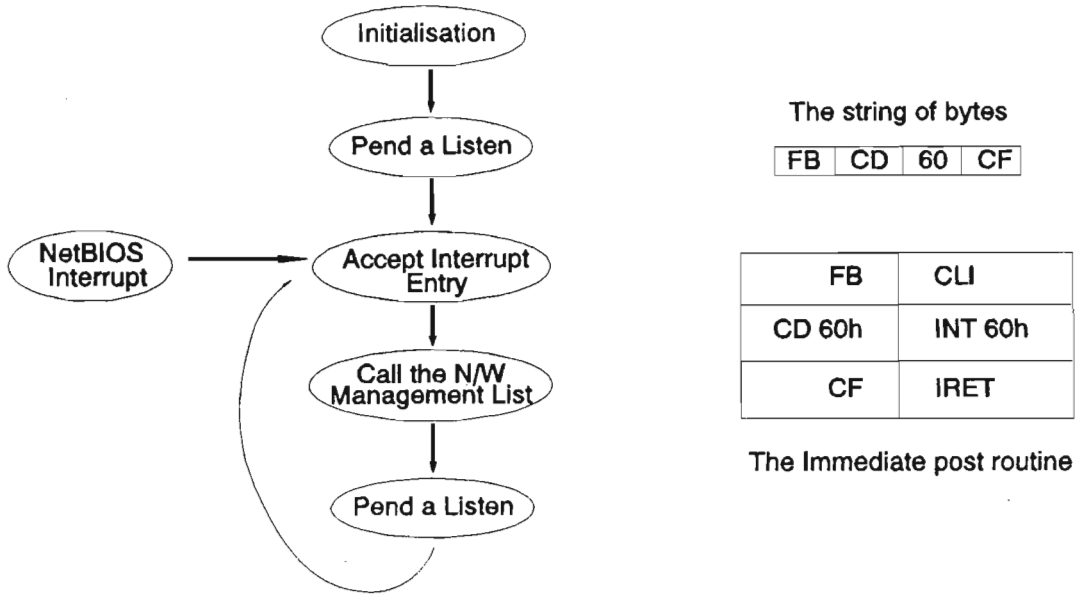

Figure 5.4 The Listen Routine Process

management task to update the network management list. The listen routine accesses the network management list to pass the session number of the new channel and the name of the partner. The call to the management task is executed outside the scope of the interrupt entry to decouple the interrupt handler. Once the update has completed a new listen command is pended and the procedure is repeated as illustrated in figure 5.4. If a remote node issues a call while the interrupt handler is dealing with a previous call and before a new listen is pended, the call will time out with a failure code and the application will be permitted to reissue the request.

Two other interesting network management functions, the *redirect* and *hangup* functions, are encompassed within a single task (with separate entries) but perform very different functions. Meridian Ada automatically allocates one kilobyte of stack space for each task (this can be altered with effort) and the use of a single task minimises the overhead.

The *hangup* function is used to close a channel and is the corresponding entry to the initiate function which opens a channel. The *initiate* function is based upon the NetBIOS call routine and is a simple procedure call that is passed a partner name and returns a failure code. When

the initiation is successful the manage network task is passed the name and session number of the new partner. If the attempt fails then the failure code is passed back through the communications interface to the application and the application is expected to act upon this.

The *hangup* function is more interesting because either application existing at the two nodes on a channel can issue the *hangup* command. The *hangup* procedure call (figure 5.5) is passed through to the NetBIOS interface from the communications package.

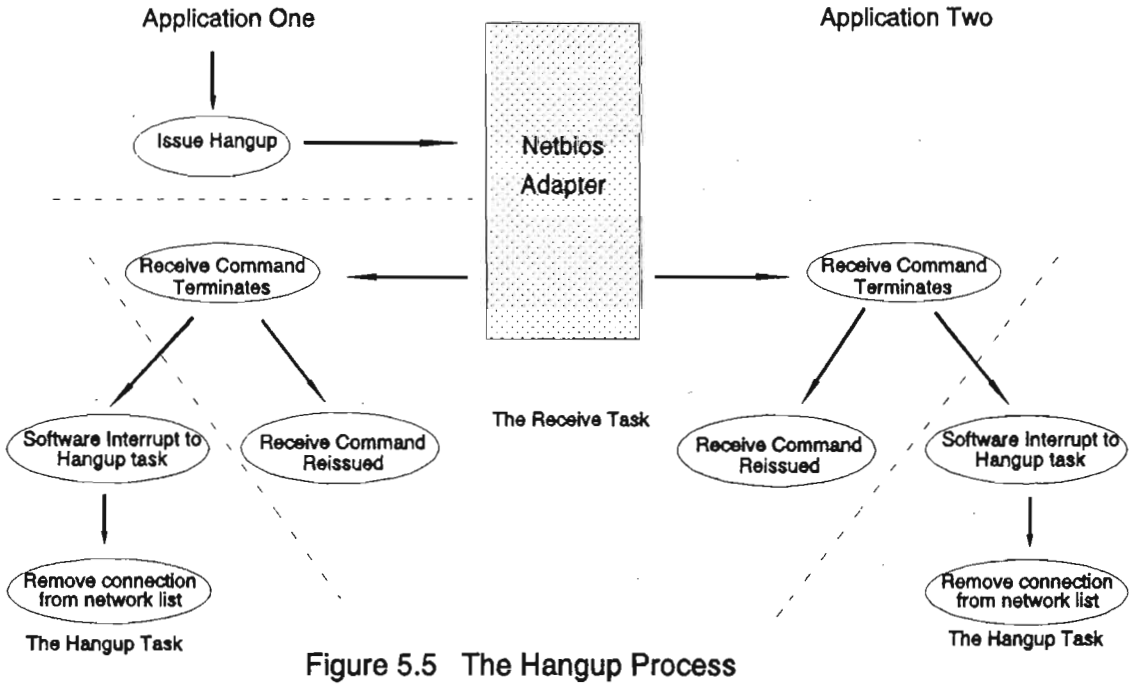

Figure 5.5 The Hangup Process

NetBIOS immediately closes any pending receive commands on both ends of the channels. However, as is discussed later, the receive command is used to receive information from any channel, not simply the one currently closed. Furthermore, at the channel end which did not issue the *hangup*, the closing of the receive (which returns a failure code and session number) is the only indication that the channel is closed and must be used to remove the channel information from the network management list. Accordingly, the receive routine traps the termination and issues a software interrupt which corresponds to an appropriate interrupt entry of the *hangup* task. It was felt that the use of another task to perform the *hangup* management was advantageous because it decouples the receiver routine, and allows it to issue a new receive command virtually immediately. Although the node that issued the *hangup* command could

explicitly remove the channel from the network list, the receive command would still terminate. The advantage of the above approach is the mechanism can be used at both ends of the channel equally effectively.

The *redirect* option allows a remote application to *redirect* the output of another application. The objective is to *redirect* streams of messages, with the minimum of disruption to the application already processing on the node, to facilitate the insertion of a new node within the logical network.

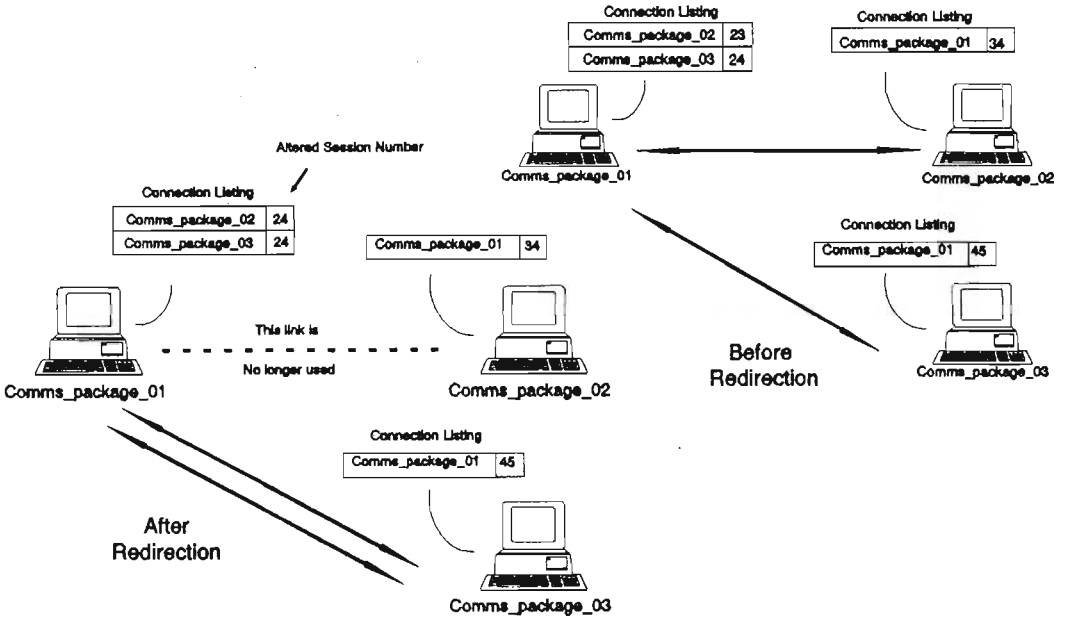

Figure 5.6 The Effect of a Redirect

Any application which intends sending a message to a target node, specifies the name and message to be sent. The sending routine obtains the session number for the channel corresponding to that name, from the network management list. The redirection operates by changing the session number that is associated with the partner name (figure 5.6). This provides a minimum of disruption to the application executing at the node and can be achieved quickly while the system is on-line. The major requirement of the new, inserted application is that the old target node, the one that used to receive the messages before redirection, continues to obtain information to allow it to continue processing. This requirement can be met since the inserted node can route the information it receives on to the original target node.

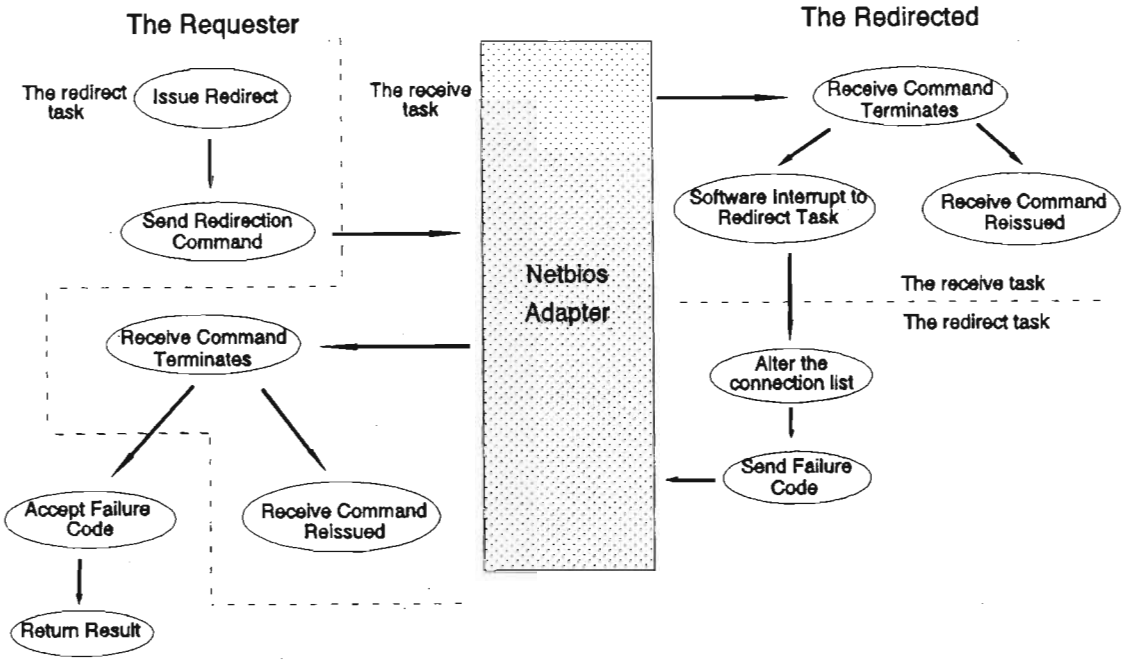

Figure 5.7 The Redirection Process

In the case of the *hangup* command, assurance of the successful completion of the command was obtained when the local receive command terminated, and the failure code from the NetBIOS *hangup* returned successfully. With the *redirect* command NetBIOS offers no such acknowledgements and this must be generated by the communications package. This requirement is met through the use of the Ada task interrupt entry. The redirection facility makes use of one task on either end of the communication channel (figure 5.7). The task at the receiving end of the channel (the redirected) is shared with the *hangup* task for the reasons outlined above. The sending task (the requester) accepts an entry from a *redirect* procedure that forms the *redirect* interface for the application. This is more convenient for the application, neater, and hides the implementation of the *redirect* from the application. The requester despatches a message to the redirected which is trapped by the receive routine. The receive routine issues a software interrupt to the *redirect/hangup* task. This decouples the receiver routine which pends a new receive command. The *redirect/hangup* task calls the *manage network list* task and alters the session numbers. The *redirect/hangup* task despatches a message to the requester acknowledging the status of the attempt. The receive task of the requester traps the return message and signals the requester's redirect task. The redirection process completes when the failure code is returned through the procedure call to the application routine.

5.2.2 The Send and Receive Functions.

The send and receive tasks are the essence of the communications system. They allow an application to despatch information to a named destination node and allow it to receive any information sent by other nodes.

As in the *redirect* and *hangup* functions, and for the same reasons, the send option is provided through a procedural interface that has entries to a send task. Since the send is an integral part of the communications system it is offered with a number of alternatives. Most important is the differentiation between the synchronous and asynchronous sends. The asynchronous send does not block the calling program but allows it to continue processing as soon as message has been copied across to the sending routine. This is useful when the application cannot afford to wait for, or does not care about any acknowledgement. The main problem associated with this option is buffering. The message needs to be stored before it is delivered and the failure code matched to a particular send and returned when the transaction has completed. Furthermore an Ada "out of storage space" exception would be raised when the temporary storage space required for the message transactions exceeds that on the heap. The design and correctness of a distributed system becomes more difficult to assess when buffering is employed since the state of the system no longer depends only upon that of the application processes but also on the information stored in the buffers. The solution the communications package employed was to allow limited asynchronous facilities (mainly for low priority messages) and provide more options for synchronous communication (which will be discussed later). The communications package limits the asynchronous send to one message at a time and retains the failure code of the last message transaction. An entry to the send task allows the application to retrieve the failure code. The message buffer is cleared before an additional asynchronous send is permitted.

The synchronous send options are more extensive. The advantage with the synchronous send is that the application receives the failure code from the send routine as soon as the routine completes. As a result any decisions made are based upon current information and the application does not need to (make an especial effort to) fetch the failure code. In addition, the implementation of the communication routine is far more efficient since there is no buffer of

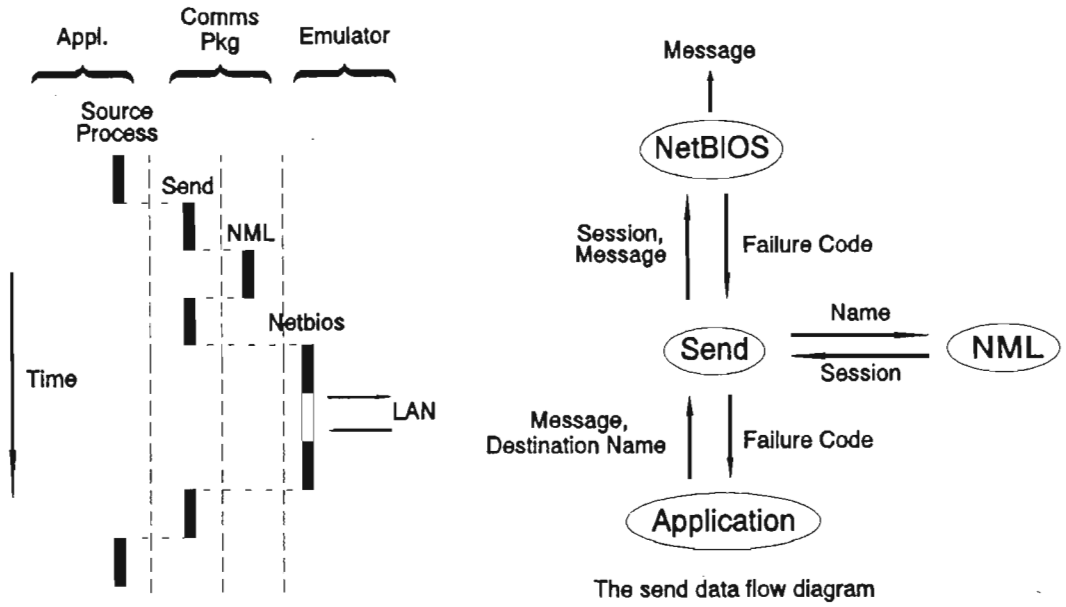

Figure 5.8 The Synchronous Send

unsent messages and failure codes that have not been fetched. The cost is the time that the application is forced to wait, without processing, for an acknowledgment signal. Figure 5.8 depicts the execution of the send against time, and a data flow diagram to indicate the flow of information once a send command is issued. The communications package offers two types of acknowledgement that are used for messages of different importance. The receiver routine is implemented using a fixed buffer size (motivated shortly) after which further messages cannot be accepted. The application can specify whether it requires full acknowledgement which indicates that the message has successfully negotiated the communications network and found space in the buffer at the target node. The alternative is communication acknowledgment. The latter is based upon the NetBIOS failure codes and indicates to the application that the message has successfully been received but not necessarily placed in the buffer (the application may have a full buffer of messages to process). In this event the message is discarded. The former alternative requires the receiver routine to send a return message to indicate that the message has been placed in the buffer or discarded. A fully acknowledged transaction is depicted in figure 5.9.

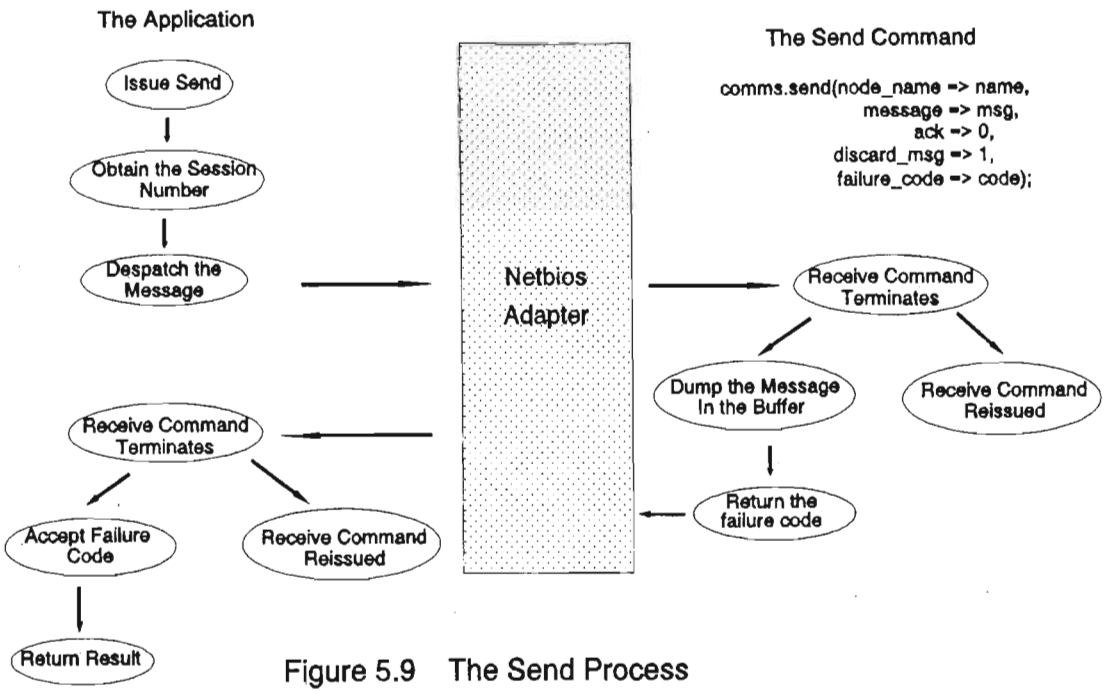

Figure 5.9 The Send Process

The communication system does not provide a multi-destination send as this may be implemented equally easily and with better control by the application package using the conventional options that are provided in a connection-oriented interface.

The receive task is the other part of the session connection. It is encapsulated in an Ada task to provide it with its own thread of execution and give the appearance of concurrency. The receive task initialises the receiver, an assembly language handler, that pends and controls the NetBIOS receive command. The receiver has been implemented in assembly code to improve its efficiency and increase its speed. The receiver pends a NetBIOS *receive_any* command with the "no-wait with interrupt" option (figure 5.10). The receiver is structured along similar lines to the listen routine (already discussed) though the problem of specifying an address for the post routine is greatly simplified, and more elegantly solved in assembly language. The assembly language interface of Meridian accepts the code produced by the Microsoft 5.1 assembler for the 8086/8088 microprocessor. This was convenient since it meant that the Meridian linker could be used to build the final program (It is possible to pre-link a Meridian program and link the final version using a conventional dos linker but the final product is considerably larger).

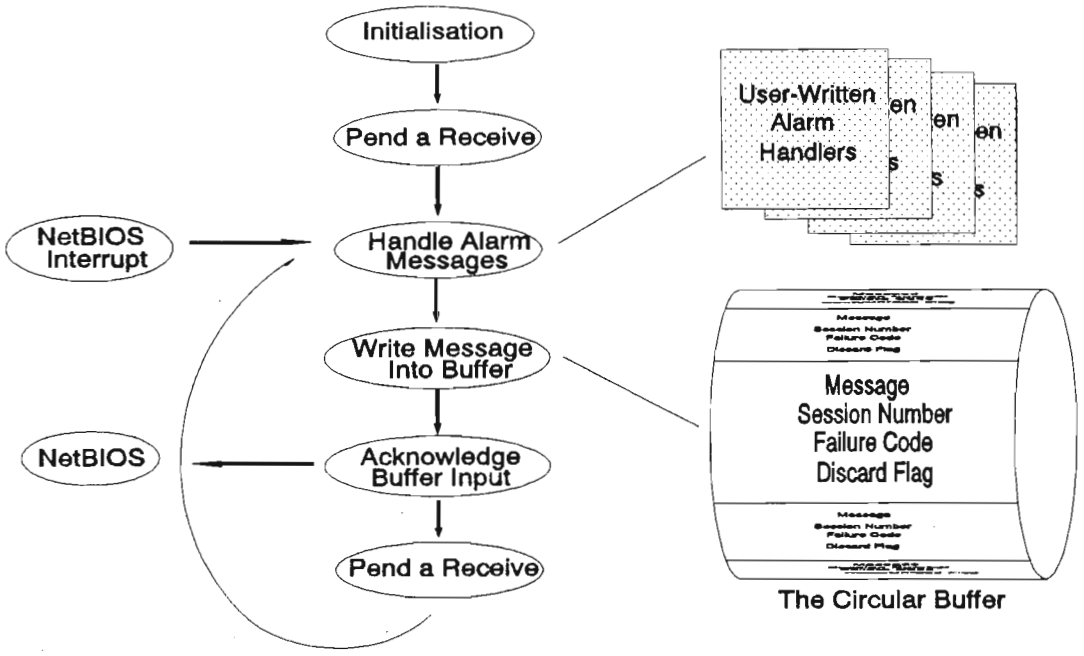

Figure 5.10 The Receive Routine Process

When a NetBIOS interrupt is received, the receiver checks to determine whether it is an emergency or special status message (to be discussed later). The receiver flow chart is shown in figure 5.11. If this is the case, execution branches to an emergency handler. Otherwise the status of the message buffer is checked to determine whether space remains for the new message. If so, the receiver dumps the newly received message, return code, session number and the read flag (discard message flag) into a circular buffer whose size is defined by the programmer when the application is designed and compiled. The storage space is allocated when the receive task is initialised. The incoming message also specifies whether it should be acknowledged or not. If so, an acknowledge message is sent (as discussed in the send section). A new receive command is pended.

A limitation of the above system is the restricted size of the circular buffer. A conventional queue has the advantage that it may be extended as long as storage exists on the heap, whereas the circular buffer has a fixed size which reduces the flexibility of the system. However, this prevents allocating and releasing memory on the heap during an interrupt routine. Furthermore, when using a queue, an error handler is required within the interrupt routine to cater for the case when storage space is exhausted during an allocation attempt. Thus the buffer may be

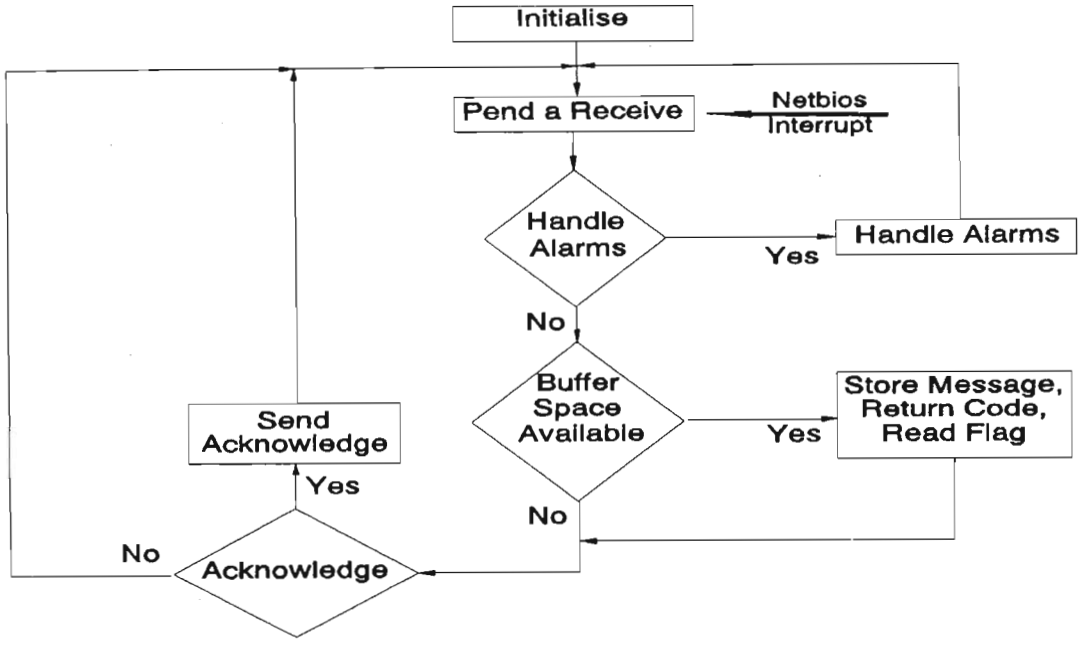

Figure 5.11 The Receiver Flow Chart

manipulated more quickly than the queue and its deficiencies are mitigated by allowing the user to specify the buffer size at each application node. The specification for the full receive buffer is included in figure 5.12.

Access to information placed in the receive buffer is achieved through the receive task. Upon initialisation the receive task passed a pointer to the receive buffer to the receiver. The receive buffer is a record with nine fields. These fields store information pertaining to the receive command:

1) *name_number : It is required by NetBIOS to issue the receive_any command.*
2) *msg_length : This is the maximum length of any message in the system and is defined by the application programmer.*
3) *return_code : This represents the return code of the first receive_any command pended - it should contain the value 255, which indicates a successful pended NetBIOS command.*
4) *buffer_length : The buffer_length is defined by the application programmer and is the maximum number of messages that may be stored at once in the receive buffer.*

```
Generic
   max_message_length : integer;
   max_buffer_length : integer;
Package receive_dfn is

   -- Structure for Individual Messages --
   subtype Message_type is string;
   type receive_record is record
       name_session_number : integer := 0;
       read_flag : integer := 0;
       return_code : integer := 0;
       message_buffer : string(1..max_message_length);
   end record;

   type received_messages is array(1..max_buffer_length) of receive_record;
   type buffer record;
   type buffer_record_pointer is access buffer_record;

   type buffer_record is record
       name_number : integer;
       msg_length : integer;
       return_code :  integer;
       buffer_length :  integer;
       msg_pntr : integer;
       session_number : integer;
       counter : integer;
       msg : message_type(1..max_message_length);
       message_all : received_messages;
   end record;

end receive_dfn;
```

Figure 5.12 The Receive Buffer Specification

5) *msg_pntr : This points to the latest message received.*

6) *session_number : This points to the latest session number.*

7) *counter : This indicates the latest message that has been read by the application.*

8) *msg : The buffer that is passed to the NetBIOS receive any command in which the message is placed by NetBIOS.*

9) *message_all : this is an array of buffer_length of records that contain the information received by the receiver for each individual message.*

The fields of the individual records contain the following information:

1) *name_session_number : the session number of the message stored in this record.*
2) *read_flag : indicates whether the message has been read, has not been read and may be overwritten by a more recent message, has not been read and may not be overwritten.*
3) *return_code : indicates the status of the receive any command that received the message stored in this record.*
4) *message_buffer : an array of bytes in which the message is stored.*

The last direct communication between the receiver and the receive task is when the receiver is initialised. The receiver and receive task share access to the common message buffer storage space. Since the receiver is an interrupt routine, and the communications package may be forced to operate under a pre-emptive scheduler, any instructions that access the shared memory must be executed within a critical region. The application requests received information through a procedural interface (as discussed previously) from the receiver task. The region where the receiver task copies information out of the message buffer is treated as a critical region and interrupts are suppressed momentarily. When the message is copied out of the buffer and is processed by the application, the read flag is set to indicate that message buffer may be overwritten. The communications interface allows an application to send a message with the proviso that if a later message arrives the old one will be overwritten, and the receiver will simply overwrite and destroy the old message. This is useful when the message content is inconsequential.

The above send and receive routines will deal adequately with the normal operation of a process control system. At times, emergencies will arise for which the usual channels cannot satisfactorily cater. The application programmer is provided with the facility to issue alarm messages which take precedence over the message buffer and the current task executing. The receiver routine will trap any message that specifies an emergency and is allocated an interrupt number by the application programmer. The receiver will generate a software interrupt for that number, and the message is placed into a special emergency buffer. The application is expected

to cater for the emergency by maintaining a task with an interrupt entry that corresponds to that number.

There are a few issues that are raised by this facility. The 8086/8088 assembly language does not permit variable software interrupts. This problem was solved, in the receiver, by accepting a standard interrupt to a prehandler (that is activated by the NetBIOS receive interrupt), which simulates an interrupt to the desired Ada interrupt handler. The Ada handler completes with the execution of an assembly language *iret* instruction. The pre-handler terminates with an *iret* to comply with the requirements for the NetBIOS interrupt option. The effect of a variable interrupt was obtained in the following manner:

1) *The interrupt address, corresponding to the interrupt requested in the message sent, was obtained from the vector table.*
2) *The flags register was pushed onto the stack.*
3) *The code segment register was pushed.*
4) *A near call was performed to the nearest possible location to push the instruction pointer onto the stack.*
5) *The offset (instruction pointer) on the stack was altered to point to the next instruction after which the "variable" interrupt would have occurred.*
6) *The interrupt address (code segment and instruction pointer) was pushed onto the stack.*
7) *A far return was performed which had the effect of an interrupt.*

This provides the application with the facility to issue an emergency interrupt from one node, that will be passed on to the application processing at the remote node. An emergency task is the application program equivalent of an interrupt handler so the *caveats* that apply to a conventional interrupt handler apply here. It is important that the application at the remote node possesses the required interrupt handler to deal with the received interrupt. A new receive command is not pended until the end of the scope of the interrupt accept statement, thus it is essential that the majority of the processing is accomplished out of the scope of the accept.

When the communication completes, the application issues a *hangup* command which closes the communication channels. Eventually, once all channels have closed, the tasks that manage the network are shut down through the interface's *reset* command. This procedure checks to determine that all channels are closed and proceeds to issue a NetBIOS reset command, which removes all trace of the node on the network.

5.3 ERROR CODES, SYSTEM LIMITATIONS AND BOUNDARY CONDITIONS

Error codes are returned from communication package calls and are based, for the most part, on those supplied by NetBIOS. However, these codes do not cover all aspects of the communication package and in these circumstances additional error codes based partly on the NetBIOS codes, where applicable, are generated to represent the specific error case.

An error code based upon a communication operation reflects a repeated attempt by the communication package and NetBIOS to establish a link, or to pass information. This environment was developed using Netware 2.15 (on an Arcnet LAN) which makes use of a shell configuration file to set options for the IPX communications protocol and NetBIOS version 3.01 emulator. These fields reflect values that define the number of sessions, buffers and the time-outs of a particular adapter and this may alter the meaning of the error codes. If a time out has been reduced to such an extent that a previous communication failed and returned an error code because the target was momentarily busy, this may justify another attempt at the transaction which may succeed. Where the networks vary, the method of specifying shell configurations may also vary but under all circumstances the user of the communication package must take cognisance of the effects of these defaults.

The communications system has been designed to provide reliable transfer of data across a local area network. The efficiency of the communications package is bounded by the network used, the size of the messages and the buffer, and the degree to which the transfer needs to be acknowledged. If a process can be designed to make use of the package in such a way that the individual nodes do not need to make use of acknowledgement codes for all messages that are sent, this will reduce the number of messages that must be transmitted across the network and

may increase the throughput and performance of the package. In certain applications, such as display updating, this is quite possible. Further, the use of alarm handlers allows the application to deal with alarm situations immediately. The misuse of this facility, however, could cause starvation of other tasks and the loss of incoming messages. Where possible these methods of dealing with problems that arise should be avoided in favour of standard messages that may be confirmed.

The efficiency of the system is reduced when the size of the messages and buffer, defined during the instantiation of the communication package, exceeds that of the actual size that is required. When the package is instantiated storage space equivalent to that of the buffer size and a small working space is reserved. This reduces the size of programs that may execute and introduces unnecessary overhead in the transfer of messages from one node to another.

The errors, both for the standard NetBIOS interface and that of the communications package are reflected in Appendix B, as are the fields for the shell configuration file that pertain to NetBIOS.

This chapter has discussed the logical structure of the communications package, some of the problems associated with its design and a few technicalities of its construction. Inter-node communication can be one of the most expensive processes in a distributed system as it is relatively time consuming and may retard the running process. Consequently it should be used as little as possible. The following two chapters will discuss two simple case studies that have used this communications system. The chapter beyond that will concern itself with the Meridian Ada compiler, NetBIOS and the MSDOS operating system and how this software has affected the environment. It will conclude with a brief discussion of the Ada9X project.

CHAPTER SIX

The Hot-Line Communication System

6.0 INTRODUCTION

The Hot-Line Communication System (HLCS) was used in the development of the communications system to test the NetBIOS interface and the basic concepts of the communications package. The HLCS is based upon a design by Shumate [31], which provides for bi-directional communication between two centres, Red and Blue, on two different nodes across a local area network. It illustrates how Ada, with its emphasis on modular software and reusability, can be used to implement the communications package which interfaces with NetBIOS to establish and maintain the distributed system. A brief, object-oriented approach is presented for the design of the HLCS.

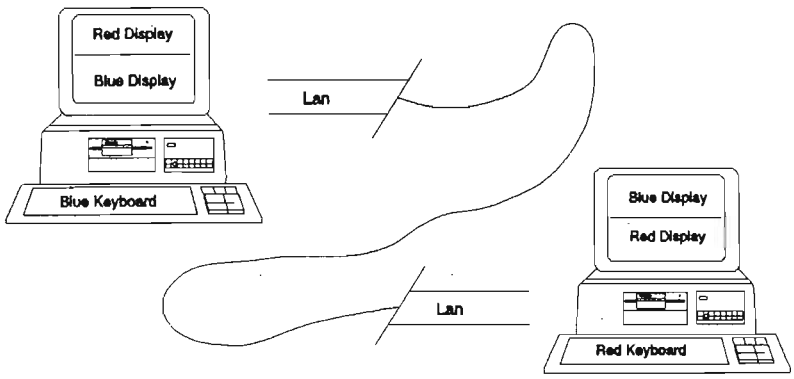

Figure 6.1 The Hot-Line Communications System

6.1 THE PROBLEM DESCRIPTION

The HLCS consists of two communication centres, Red and Blue. The user at the Red system types one line messages which are limited to a maximum of 64 characters at the Red_Keyboard (Keyboard refers to the Ada Keyboard task rather than the actual physical keyboard or the ROM keyboard handler). If there are fewer than 64 characters the message is terminated by a carriage return. At each node there is a Red_Display, a Blue_Display and one keyboard. Messages typed at the Blue_Keyboard will be echoed at the Blue_display of the Blue node and will be sent to the Blue_Display of the Red node. The composition of a message may be interrupted between characters to display a received message from the remote node. The same

applies to the Red_Keyboard and in this manner bidirectional communication is achieved (refer to figure 6.1).

The hardware controlling the display and the keyboard is of little interest since routines are provided within the Ada environment as interfaces to these devices.

6.2 TOP-LEVEL DESIGN

At this level emphasis is based upon object-oriented system decomposition, on the processes that occur simultaneously. The HLCS uses a task for each external device to implement those processes that should run concurrently [32]. These tasks are used as the interface to the keyboard handler, the display driver and the relay; the relay calls the communications interface. The communications package provides a set of Ada routines to manipulate NetBIOS.

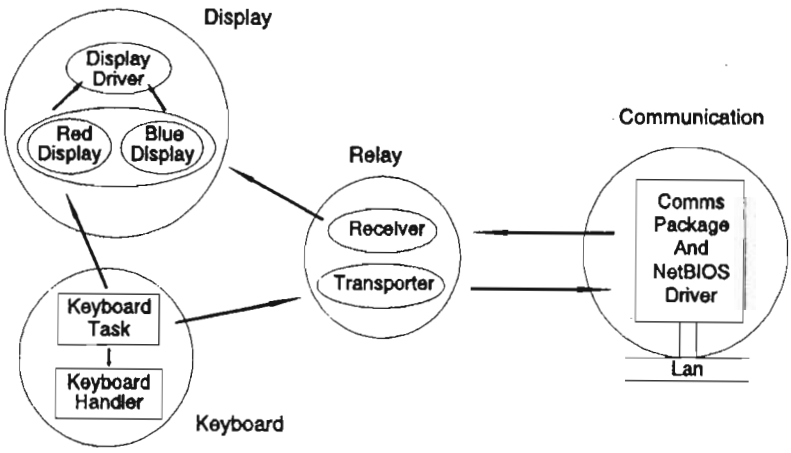

Figure 6.2 An Object Oriented View

Figure 6.2 gives an object-oriented view of the system. The Display, Keyboard, Relay and Communications package are all seen as objects. Characters are passed between the objects, either as single characters or as a number of characters forming a message. The operations on characters are defined by the objects and are hidden from the other objects within the system. Data abstraction prevents inadvertent modification of internal data types of the objects. This approach of data abstraction and the restriction on the visibility of operations within the objects, allows the modules of the HLCS to be developed and tested independently. For example, the

communications package may easily be replaced with a serial line driver with very minor changes to the Relay object.

Only the Display object, written in Ada, is visible to the both of the other objects within the system. The Keyboard object interacts with the same colour display, within the Display object and the Relay, to dispatch completed messages and the keyboard handler. It is never required to handle calls from either the display, the relay or the keyboard handler. After the message is built the Keyboard makes a call to deliver the message to the Relay object which calls the communications system. This package is responsible for delivering the message, through NetBIOS, to the relay and display at the other end of the communications channel (refer to figure 6.3).

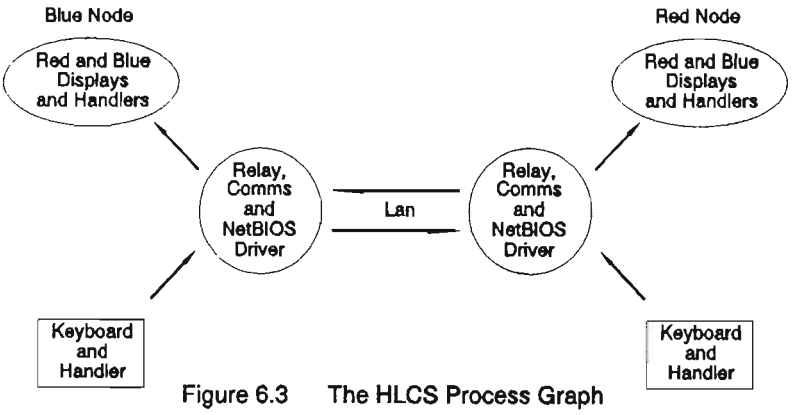

Figure 6.3 The HLCS Process Graph

The keyboard cycle is described by:

```
loop outer_cycle_for_keyboard_task
        loop inner_cycle_to_get_a_message
                check_to_see_if_character_pending;
                        get_a_character;
                        display_the_character;
                        add_the_character_to_the_message;
                task_switch_directive;
                exit_when_the_message_is_complete;
        end loop inner_cycle_to_get_a_message;
        send the message to the other display through the relay and communications interface;
end loop outer_cycle_for_keyboard_task;
```

The Display object has an important characteristic that differs from the Keyboard. If neither a message is pending nor the operator typing a new message, the Display must be ready to

accept either a message from the remote node through the Relay, or a character from the Keyboard. It is this characteristic of interacting non-deterministically with either of two other objects that makes it preferable for the Red_Display and Blue_Display, forming the Display object, to be designed as called tasks (except for their calls to the display handler to output the character).

This closely coupled interaction raises a potential problem. If the operator is busy typing in a message, the other colour display might be blocked while the Keyboard/Display interaction continues. This problem was solved through the buffering facility of the communications package for incoming messages and forcing a task switch in the keyboard task after checking whether a character is pending in the keyboard buffer. The relay/communications package interaction has uncoupled the Display, though the caller/called relationship has been maintained. If the communications package is replaced by another driver (e.g. serial) the incoming messages may have to be buffered within the relay.

6.2.1 Mini Specifications for the Processing of Blue Messages

Blue_Keyboard

The Blue_Keyboard obtains characters from the keyboard handler, sending them to the Blue_Display as they are received. It builds a message from the characters, sending complete messages to the Relay when ready. A carriage return indicates a complete message. The carriage return is sent to the display and inserted in the message sent to the Relay.

The Keyboard does not check the characters received nor does it ensure that a carriage return is received before the end of a line. The Keyboard does not block while waiting for a character or putting a message into the Relay, or character into the Display.

Relay

The Relay accepts messages from the Keyboard and delivers them through the network to the other Display. The Relay consists of two threads of control, a transporter and a receiver. The transporter accepts messages from the keyboard and despatches them through the

communications channel to the receiver at the remote node. The receiver calls the Display at the remote node.

Blue_Display

The Blue_Display accepts either messages from the Relay or a character from the Blue_Keyboard. Preference is given to the display of characters from the Keyboard. Once a message is received the entire message is displayed, before any further consideration is given to displaying another character from the Blue_keyboard. The display treats ascii line feeds and carriage returns as standard characters.

There is an implicit requirement that the messages be received in the order sent (i.e. a first in first out queue).

6.3 THE HLCS COMMUNICATIONS INTERFACE

```
If comms.check_for_netbios then
    comms.system_setup(failure_code => code,
                       retry_count => retry_count);
    If code=0 then
        comms.get_connection_array(name => name_array,
                                   lngth => array_length,
                                   code => new_code);
        ......
        ......
        ......
    else
        tty.put_line(" Communication Initialisation Failed");
    end if;
else
    tty.put_line(" NetBIOS is not installed");
end if;
```

Figure 6.4 The Initialisation Procedure

The generic communications package provides the Relay object with an interface to the local area network. The parameters that are passed through to the communications package when it is instantiated represent the size of each message and that of the buffer. A buffer size of one messages for a sixty-four byte message length was selected.

The message size is based upon that used by Shumate in [30] and the buffer length is not crucial in this instance (it could most likely have been significantly shorter - e.g. two

messages). Before the Keyboard object becomes active the Relay checks to see whether NetBIOS is present and activates an initialisation routine in the communications package. This routine requires a retry count to specify how many times it should attempt to initiate communications with a partner and returns a failure code to indicate the result. If the transaction was successful then the name of the connection partner is obtained for future communication transactions. The relay enters a loop waiting until a valid partner is returned. Once the initialisation routines have completed then the system is ready to send and receive messages.

```
comms.send(node_name => partner_name,
                message => message,
            discard_msg => 1,
                    ack => 0,
           failure_code => code);


comms.get_latest_message(message => message,
                         msg_pntr => message_pointer,
                             name => sending_name,
                    failure_code => code);
```

Figure 6.5 The Message Transfer Instructions

The relay object consists of receiver and transporter components. The receiver component obtains the messages that are sent by the partner. The communications package returns the message, the position of the message within the buffer, the name of the partner and a failure code to indicate the result of the transaction. The display object is called when the failure code indicates that a message has been successfully received.

```
comms.hang_up(name => partner_name,
              code => code);
.....
if code = 0 then
   tty.put_line("Adapter has been closed");
end if;
comms.reset(failure_code => code);
if code = 0 then
   tty.put_line("Adapter is reset");
end if;
```

Figure 6.6 The Termination Procedure

The transporter routines despatch the message that is created in the keyboard task. The partner name is obtained during the initialisation procedure. For this application, the send command

specifies to the communications package that it requires the messages to be acknowledged and that they may not be overwritten in the buffer until they have been read by the remote application. A failure code indicates the status of the transaction.

Finally when the communication between the two nodes is complete the Keyboard task accepts an escape character as a termination directive. The communications package is instructed to hang_up the connection and to reset the NetBIOS adapter. The Keyboard, Relay and Display objects are terminated and communication ceases.

6.4 CONCLUSION

The HLCS requirement is typical of a small real-time system. Taking an object-oriented approach results in a straightforward design that is easily understood and implemented. The most interesting requirement is that of the display. It must be prepared to display either a single character or a message at any time (i.e. interact non-deterministically). It must not allow the creation of a message to be blocked and must continuously display incoming messages.

The HLCS is a simple example using the basic operations provided by the communications interface. It has illustrated that a communications program using the interface is feasible, easy to implement and operate, and provides a sound basis on which to build more complicated distributed message passing systems. The following chapter deals with a simple control system to manage a set of pumps and provides a further test of the communications package.

CHAPTER SEVEN

A Simple Pump Control System

7.0 INTRODUCTION

This simple pump control system was used to further test the communications package. The pump control system is based upon work done by two senior Electronic Engineering students for their final-year projects [33,34]. This chapter will briefly review the functional requirements of the pumping system and discuss the structure of the code and the use of the communications package. Although the processing load placed upon each computer is low and the control algorithm could be implemented using one computer, the objective is to establish the viability of the communications package in a simple process application.

7.1 THE PROBLEM DESCRIPTION

The pumping system consists of a single valve, two water tanks and pumps that are controlled using two computers connected via the communication package and the local area network. A further computer functions as a user friendly interface and allows the user to impose set points and track the water levels in both tanks (figure 7.1).

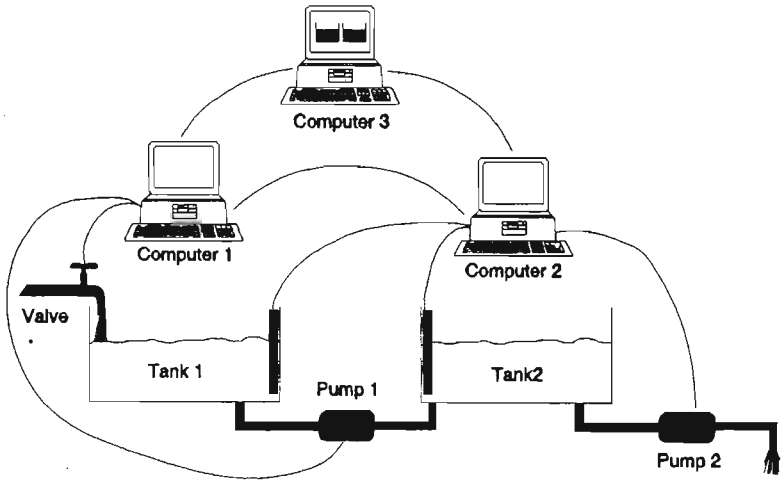

Figure 7.1 The Pumping System

The third computer communicates the set points to the controlling computers. Two computers read the tank levels, control the pumps and the valve and communicate with one another in

order to maintain the depth of the water within a threshold value of the set point. The state of the system is periodically despatched to the user interface computer.

An analogue to digital converter which reads the depth of the water is polled every one tenth of a second. The pumps and valves are controlled in an on/off fashion, through the printer port. Two alarms representing the maximum and minimum levels are set for each tank, and raise an emergency signal if exceeded.

7.2 TOP-LEVEL DESIGN

As in the previous case study, the system will be decomposed in terms of the functions that are performed. Initially the system may be resolved into three physical nodes that cater for process control devices and a graphical user interface. The functions that execute concurrently are encapsulated within Ada tasks. The applications communicate across the network with one another using the communications harness developed, although the control system could be easily modified to use serial communication across an RS232 cable as a replacement. The following sections take an object-oriented approach and briefly discuss each of the three physical nodes of the system.

7.2.1 The Code Structure of Computer One

Computer One is used to control a pump and to obtain the depth readings of the two tanks (figure 7.2). The level object polls an analogue to digital converter that returns a value proportional to the level of the water in the tanks and computes the percentage of full capacity. The control object obtains the water depths from the level object in order to set the state of pump one, and to despatch the information to the other nodes through the communication interface. The set point and tolerance bound is obtained by decoding messages received from node three. The pump controller receives a boolean signal from the control object to specify the state of the pump. Both pumps are AC pumps and are controlled in an on/off fashion. The flow chart for computer one is included as an example (figure 7.3).

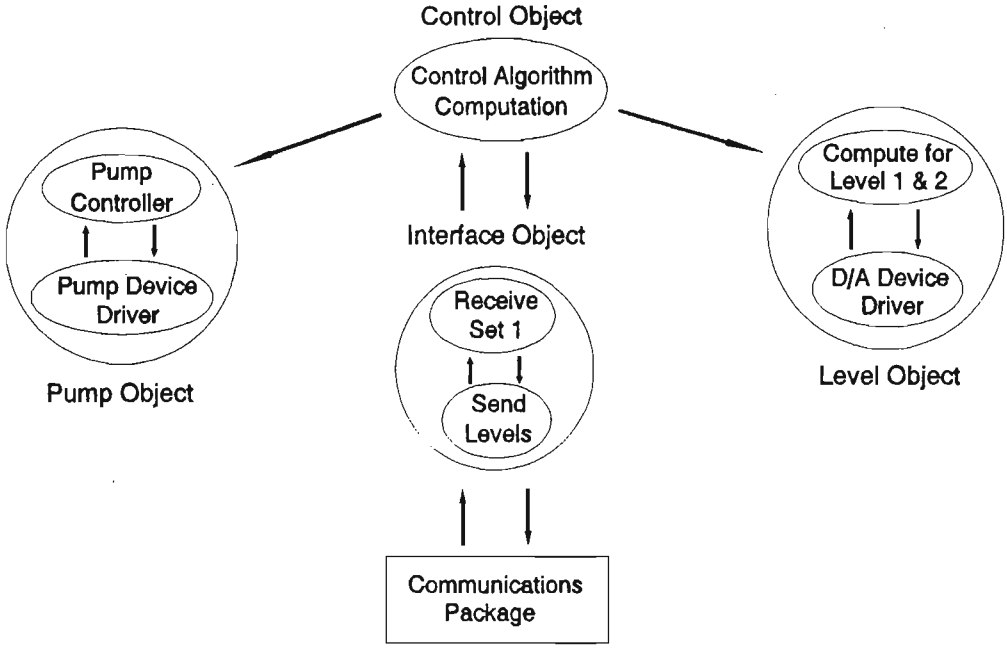

Figure 7.2 The Object-Oriented View of Computer One

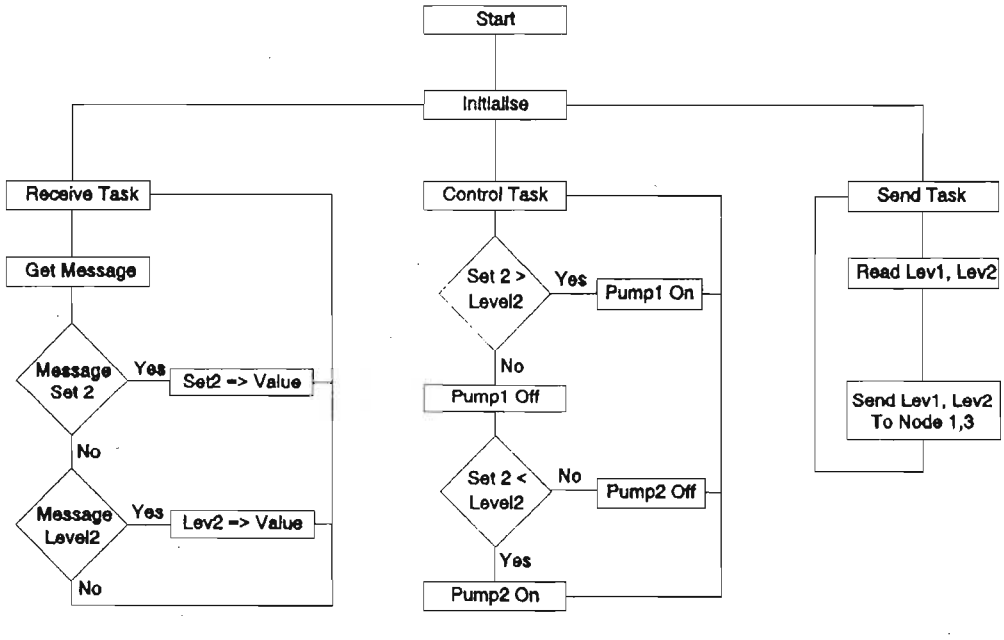

Figure 7.3 A Flow Model for Node One

7.2.2 The Code Structure of Computer Two

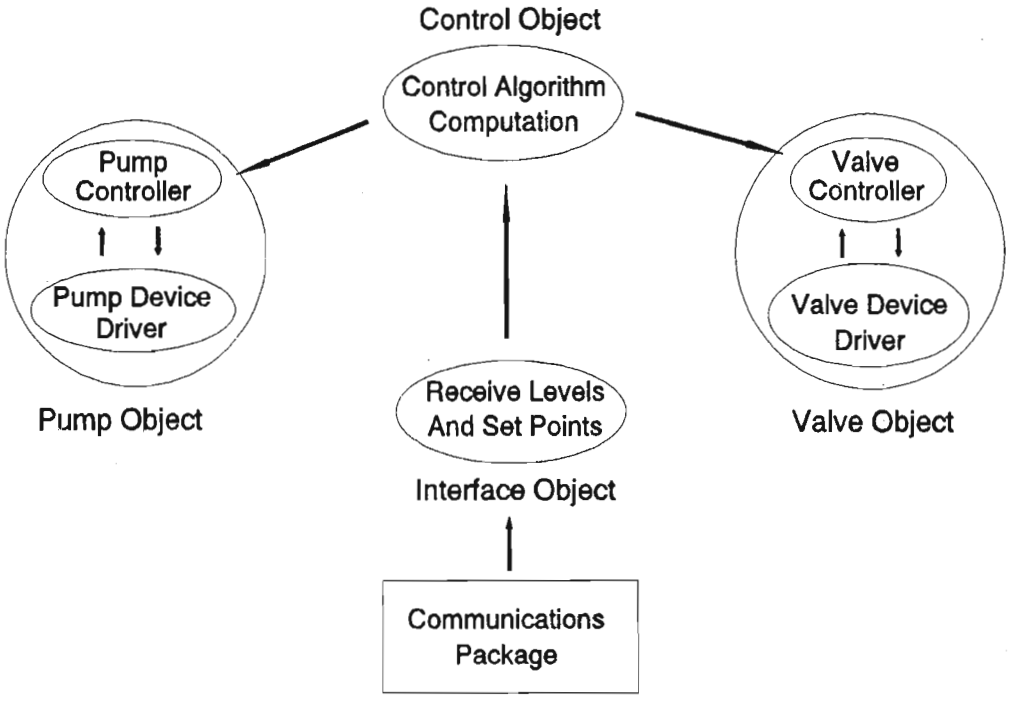

Figure 7.4 The Object-Oriented View of Computer Two

The pump, control, valve and communications routines are all seen as objects (figure 7.4). Since both the pump and valve are operated in an on/off fashion the control object passes boolean information specifying the required state of the devices. The interface object obtains the set point, current water level and the tolerance bound of tank two, through decoding the incoming messages received from the other nodes. Corresponding action is taken to either open the valve or start the pump to correct the water level.

7.2.3 The Code Structure of the Graphical User Interface

Node three provides a user friendly interface for setting the water level set points and tolerance bounds, and for observing the state of the system (figure 7.5). The user interface allows the operator to alter the desired tank levels and set a threshold to prevent an inordinate amount of switching of the valve and water pumps around the set point. The console object does not

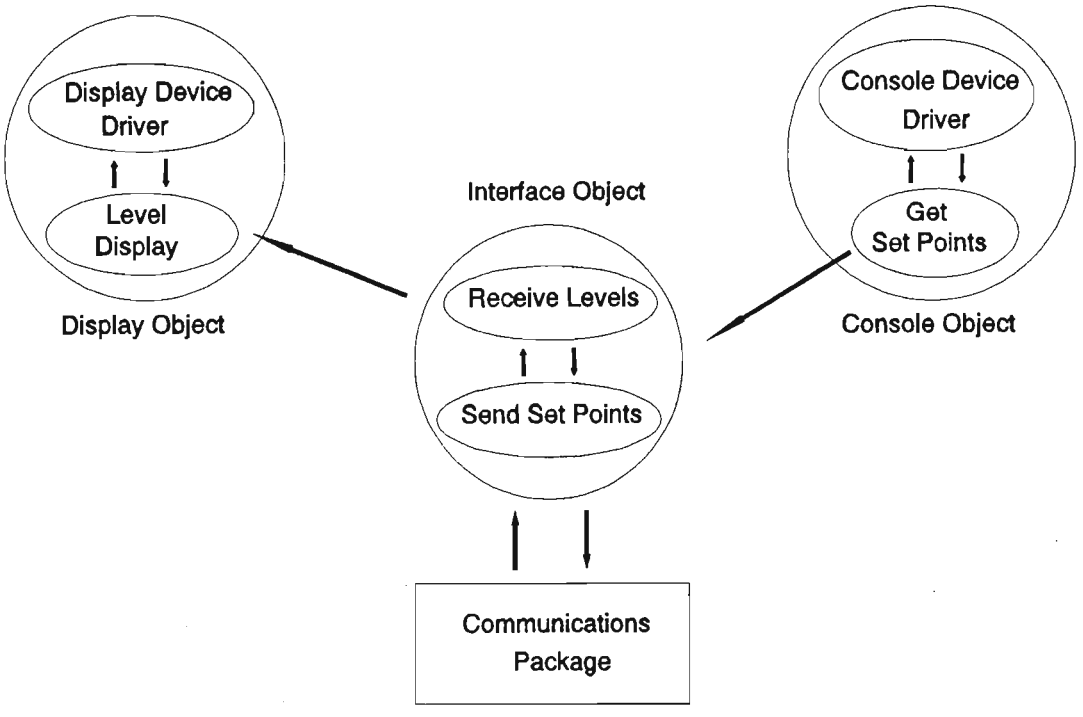

Figure 7.5 The Object-Oriented View of Computer Three

suspend the operation of the other tasks in the system. The display object allows the operator to track the levels in the tanks and displays the state of the pumps and valve.

7.3 THE COMMUNICATIONS INTERFACE

The generic communications package provides the interface object with access to the local area network. Since the water levels were sampled every tenth of a second it was felt that a buffer size of five messages for a twenty byte message would be adequate. The message size was quite sufficient to convey all the necessary information required. The initialisation, send and *get_latest_message* routine code is not discussed here since it is almost identical to that of the previous case study. The code is included in the appendices for further reference.

The graphical user interface permits the user to halt the monitoring node without suspending the two controlling processors. The last set point and tolerance will be maintained until another monitor is brought on line. This new monitor can seize control through the communications

redirection facility. The redirect function call specifies the target (the first or second node) and the redirected (monitor) node. The new node issues a redirect command to the target and then receives the information the target used to send to the old monitor node. The old monitor node can then perform a *hangup* command and close the communication channel with the target. If the redirection is unsuccessful the old monitor can still send new data to the target. If the new monitor issues a *hangup* command the target begins to send information to the old monitor. If no monitor exists the information is discarded.

7.4 CONCLUSION

The simple pumping system is typical of a small real-time process control system. An object-oriented approach results in a system design that is easily understood and implemented. The most interesting operation is the redirection of messages from one target to another. The redirection was found to work satisfactorily and provided a method of disengaging one processor while engaging another. The pumping system has illustrated that the communications package can be used to implement a small control system effectively.

CHAPTER EIGHT

Performance Considerations

8.0 INTRODUCTION

The two previous chapters have considered simple applications using the communications facility. This chapter considers factors which have affected the performance of the environment and the suitability of the language, compiler, and underlying communication protocol. The final section of the chapter will discuss a few of the issues being considered by the Ada9X review committees.

8.1 THE MERIDIAN COMPILER

The communications system was implemented using the Meridian Ada compiler version 4.1.1 for the personal computer. To allow more memory for symbol space and for the compilation of realistic package sizes to succeed, the extended compiler, with extended memory was necessary.

Where the Ada Language reference manual [1] provides no strict guidelines for compiler vendors, the variations between implementations may have significant effect on the run-time performance of the code. Thus, while there is an Ada standard that is defined and that compiler vendors must adhere to, in some cases the portability of the code is subject to the details of the actual implementation.

No experience was obtained with other Ada compilers for the PC as none were available to the author.

8.1.1 Generics

The Meridian implementation of generics forces the code to expand fully each time the generic is instantiated (sometimes called a macro implementation) [27]. While this approach tends to

be time efficient it implies that code, common to more than one procedure, should be placed outside the generic, to reduce the amount of overall code that is generated. Thus, while the generic procedure may be conducive to a logical and clear understanding of the code, it could be detrimental to program size. Furthermore, the use of generics affects the dependencies of package specifications and their bodies since the compilation unit that instantiates a generic depends upon the specification of the generic and its body. In cases where significant alterations are to be performed within a generic body this can mean time-consuming recompilation of all the dependent code which would not otherwise have been the case. The communications package and the receiver buffer package were made generic to allow variable buffer and message sizes to be instantiated, by the user, when compiling the application code. For the reasons given above, this is the only place they were used.

8.1.2 Task Scheduling

The use of tasks for concurrent processes has been discussed in a previous chapters and is one of the strengths of Ada. The communications environment logically comprises a set of concurrent processes which map well onto the Ada tasking model, making its use a logical choice. Although the system could have been developed without tasks, the result would have been more complex, less structured code.

When designing an application to operate in a real-time fashion, it is necessary to select a run-time system whose task scheduler will provide the required performance. There are two distinct task scheduling methods for Meridian Ada, the pre-emptive option, and the default which is non pre-emptive. Pre-emptive scheduling allows quicker servicing of interrupts on delay timeouts, but, according to the documentation has a slightly negative effect on program performance in general [25].

Non Pre-emptive Tasking

The non pre-emptive task scheduler uses a round robin prioritised scheduling algorithm that switches tasks at activations, entry calls, completions and wait conditions. Tasks that are in infinite loops may inhibit task switching altogether; this may be prevented through the use of

a *delay one_clock_tick* (standard Ada, *delay 0.0* for Meridian Ada achieves a task switch without the actual time delay) instruction which forces a task switch. Any other tasks of equal or greater priority will run first.

Pre-Emptive Tasking

Task pre-emption is the ability to switch from a lower priority task to that of a higher priority as soon as the latter is ready to execute. The higher priority task may become ready owing to the expiration of a delay, or in particular, at the occurrence of an interrupt for which the task has a defined entry. The design of application code that runs under pre-emptive tasking is more complex.

An example illustrating the pre-emption mechanism is offered in the documentation [25] as follows: assume two tasks, A and B, with low and high priorities respectively, both access a variable count. Suppose that A is executing the statement

```
count := count + 1;
```

and has just loaded the value of count into a temporary register and incremented it, but not stored the new value into count when task B pre-empts task A. In the course of execution, task B also manipulates the variable count. Let us suppose it sets it to some value. The following then occurs:

1) *The execution of the Task A would be suspended*
2) *Task A's execution context is saved*
3) *The run-time would restore the execution context of Task B*
4) *Task B would resume execution and continue until another task switch*
5) *During execution, Task B sets count to some value*
6) *At some point Task A would be the next to run*
7) *Task A's context would be restored and execution begun again*
8) *Task A stores its new value into count*

At this point, Task A's updated value of count would be stored into the variable, and the value stored by task B, would be lost.

From this example we see that access to a shared variable may not be interrupted by pre-emption. To cater for this Meridian provides a package, named *task_control* (see figure 8.1), to control pre-emption at run-time while accessing global variables and other critical regions. An alternative method is to force tasks to access global data through a guard task. This is an effective method for controlling access to shared resources, although it introduces the overhead of an additional task switch and the task overhead itself.

```
Package Task_Control is
    Procedure pre_emption_off;
    Procedure pre_emption_on;
    Procedure set_time_slice(quota : duration);
End Task_Control;
```

Figure 8.1 The Task_Control Package

The Ada language reference manual does not specify what ought to occur when two tasks of equal priority are ready to execute. The question of which task is to run first and for how long is undefined. Time slicing is a possible approach which allows the user to dictate to the *task_control* package and the run-time scheduler how long a task may execute. The time slice, as implemented by Meridian Ada applies to all tasks and priorities. The actual time that a task may have for execution can be less than that allocated to it by the time slice directive for the following reasons:

1) *Operating system delays*
2) *Differences when converting the duration of the time slice to that of clock ticks*
3) *The disabling of task pre-emption*

The Meridian Ada run-time scheduler prevents pre-emption when it is running, in order to maintain its integrity. The DOS operating system and its associated BIOS are generally not re-entrant and a request issued to DOS must be allowed to complete before another request is generated. With full pre-emption it would be possible for this condition to be violated causing unpredictable behaviour.

Priority

For pre-emption to occur it is obvious that there must be some way of distinguishing the priority of one task over that of another. The mechanism that Ada offers here is the *Pragma Priority*. The priority of a task is fixed at compile time and may not be changed dynamically (this is being reviewed for Ada9X). The Language Reference Manual does not specify that priorities must be defined although where the priority of one task is defined, the others remain undefined. (In this version of Meridian Ada an undefined task becomes the lowest priority task). The interrupt entry is defined as the highest priority task.

There is a major problem, known as priority inversion, with the current priority model (that is under revision) when priorities for tasks have been specified [20]. If three tasks, "High", "Medium" and "Low" are given corresponding priorities and Low has mutually exclusive access to a resource, when High attempts to gain control of the resource, High is correctly blocked from so doing because Low is making use of the resource. If Medium becomes eligible for execution, since it is not intent upon accessing the resource, it (correctly) pre-empts Low as it has a higher priority then Low. Thus Medium has pre-empted High, resulting in priority inversion.

The limitations of the Ada run-time model has led John Barnes [6] to declare that "Synchronisation should be done with the rendezvous and priorities should be avoided except for fine tuning of responsiveness", and this has been heeded in the design of the communications package. Furthermore, later program maintenance may frequently result in a change of priorities because of different real-time requirements, making the use of priorities to control the run-time behaviour risky.

The communications package was designed to eliminate polling and implement the receiving functions of the communications layer with interrupt entries. These tasks are automatically defined as high priority tasks and will execute as soon as the scheduler is invoked at the next task switch. The despatching and management tasks are undefined and thus are equal in priority to the other tasks except those that are interrupt driven. If synchronisation is effected through task entries, as is advocated, then the order of scheduling can be carefully controlled in the

application. Furthermore, if the time slicing mechanism (offered by Meridian) were to be used, the size of the window would need to be specified in the communications package but it would effect the windows of all tasks executing in the application program. The size of the window may not be suitable for the application since, for example, a large window will reduce the responsiveness of the controller and a small window would introduce extra task switching overhead. The additional complications in programming a pre-emptive scheduler have already been discussed.

8.1.3 Additional Considerations

Standard Ada features that are mandated in the Ada Reference Manual are implemented differently in different compilers and this can have a marked effect on the run-time performance of the application program. Consequently, when programming a real-time application an understanding of the method of implementation of that particular compiler is necessary for optimum performance. An example of this is the *text_io* package. When programming for reliability it is necessary to guarantee that storage space is available when making a dynamic allocation. It is not always obvious that the *text_io* routines almost invariably make use of the heap, and dynamically allocate memory which is later released. The results can be catastrophic if this is not carefully considered and the heap is exhausted during a *text_io* operation.

Ada exceptions have been implemented within the communications package to deal with unexpected run-time errors. While the typical goal of the compiler vendor is to introduce no overhead until the exception is raised, there are basic trade off's when implementing exceptions that cannot always be avoided. A choice must be made between lower subprogram call overhead or faster exception handler location. The call overhead can be significant and includes the costs of reclaiming storage, popping activation records from the stack, and restoring the non-local reference environment. The communications and applications packages must make minimum use of exceptions to deal with unexpected emergencies only. The program code should cater for most error conditions arising from the process it performs.

8.2 NETBIOS

The performance of the NetBIOS protocol was extremely difficult to isolate from the performance of the code and that of the local area network. The issue is further complicated through the use of NetBIOS emulators and the problem then becomes that of comparison between one emulator and another, and their underlying protocols.

For these reasons it is perhaps more meaningful to comment upon the ease of programming the interface, rather than the sheer performance of NetBIOS, which may vary extensively under differing conditions. The interface provides easy initialisation and session construction facilities. The use of sessions allows larger messages to be sent although it imposes a heavier overhead. When programming the interrupt driven procedures it would have been most useful if the interface allowed the user to provide an alternative error message code, during the interrupt procedure, which would appear at the remote end of the session. This would have reduced the number of messages to the sending application that were required to confirm the availability of storage space in the message receive buffer. Furthermore, although the NetBIOS documentation specifies that the destruction of a session at the end of a time out is needed to ensure data integrity, it is inconvenient since a new session must be established each time this occurs. An alternative approach would be desirable. These facilities would have reduced the effort in using NetBIOS and improved the general performance of the environment.

With the acquisition of TCP/IP drivers for the Electronic Engineering Department's Arcnet, this set of protocols would require close investigation in the design of a new communications package.

8.3 MSDOS

The use of MSDOS has profound implications upon the communications package, the real-time performance of the application and the code generated by the compiler. Since MSDOS is not re-entrant this complicates pre-emption and the efficient use of interrupt handlers. An ideal

operating system will view each Ada task as a separate thread of execution. In the event of any

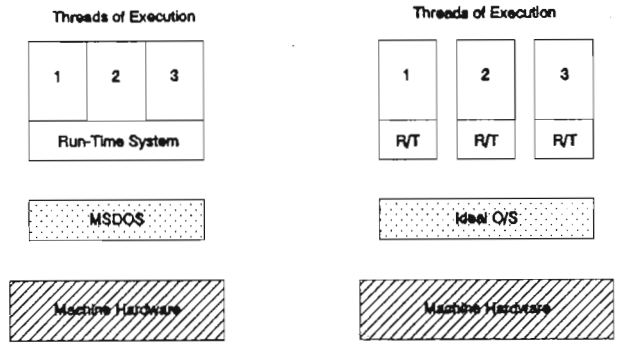

Figure 8.2 Operating Systems/ Threads of Execution

blocked thread, the execution of the program as a whole is not blocked and the other processes continue. The use of an operating system which is not real-time (as MSDOS is not) causes any subsequent program not to be real-time, no matter how efficient its run-time system. Software running under MSDOS may perform real-time applications where the time constraints are sufficiently lax, such that the performance of the operating system does not negatively impact upon the real-time system.

8.4 THE ADA9X REVISION EFFORT

The Ada9X program is a result of technical experience gained through projects since the inception of the Ada standard in 1983. According to the Ada board a revision was necessitated because "some omissions, limitations, and minor errors have been discovered ... since the ANSI standard was approved. Although a major revision does not appear to be necessary to correct these problems, some revision is warranted." [35] This section is not an attempt to cover the full scope of Ada9X but addresses some of the issues motivating for a reappraisal of the language, and to discuss some of the requirements of Ada9X relating to distributed execution. The current (the most recent documentation available to the author) movement in this area is briefly discussed.

Several key issues that affect the scope of the revision have been raised by the Ada programming community and have needed to be considered during the revision of the language.

Dissimilar Needs for Change. Users from different application backgrounds require changes for their specific applications that may prejudice other users.

Expanding Ada's Domain of Use. There is a strong feeling that if Ada is not enhanced in some aspects (distributed systems, object-oriented programming, information systems) it will fail to attract new users.

Extent of Change. Many Ada vendors fear that if Ada9X incorporates too many changes the Ada9X compilers will be less efficient, less reliable, and more costly than the existing compilers. Extensive changes may result in clients reducing or delaying their commitment to Ada until the language stabilises.

As a result a group known as the "Requirements Team" [36] suggested that those requirements could be best met through *annexes* in the 9X standard to address specific application needs. The annexes ought to be limited to the following kinds of specifications:

Specialised Packages. Standard packages are often an appropriate method of providing special facilities in certain application areas. Standardising the interface for such annexes could facilitate Ada's ability to meet specialised needs.

Pragmas and Attributes. In Ada83 only a basic set of pragmas and attributes are defined. In cases where a special pragma or attribute is appropriate then a relevant annex will provide the exact form and meaning.

Representation clauses. It is expected that annexes will provide extra restrictions on representation clause support required for the application area addressed by the annex.

Performance Specifications. Since inefficient support for certain facilities is useless under certain applications, the annexes are expected to address critical performance issues that need not be satisfied by all implementations.

The Ada9X annexes are to allow the standard to address specialised needs without mandating compiler vendors to support all annexes. This will increase the overall degree of uniformity for the annexes will impose restrictions in specific areas where none existed in the original standard. Furthermore, this allows implementors to concentrate on certain areas of implementation that are of greater interest to their customers. The following sections will deal with some aspects of the Ada9X requirements.

8.4.1 Real-Time Requirements.

Time.

Real-time systems need to measure the time of day to interface with human operators, processors and devices; they also require measurements of the elapsed time from one event to another. Ada83 offered a *time_of_day* facility but Ada9X is also expected to offer a method by which to allow periodic execution of a sequence of statements with respect to elapsed time. The current *delay* statement is not sufficient for these purposes since it may be pre-empted and lead to uncertain execution times. In addition, real-time programmers often need to specify what ought to occur in the case of a deadline being missed.

Task Deadlines.

Ada9X is expected to allow real-time programmers to specify the scheduling algorithm used for a particular program. Many users favour low-level primitives or a standard run-time interface from which scheduling policies and algorithms may be built.

A further requirement is that Ada9X supply real-time paradigms that can be used with predictably efficient performance. The paradigms to be considered are controlling access to data shared amongst different tasks, resetting task priorities in response to events, critical regions and asynchronous communication.

Asynchronous Control of Execution.

There are many situations when it is useful to terminate one procedure and start another based upon some asynchronous event (an external interrupt call or a call from another task). These situations often arise under conditions where they must be dealt with within strict time

constraints, however the need for immediate response must be balanced against that of safe and predictable execution.

Asynchronous Communication.
The ability to send a message without waiting for it to be received is common within real-time systems. Device drivers need to send data to tasks without any interrupts being missed. Agent tasks are a standard method for dealing with this situation. The overhead involved with agent tasks would reduce the efficiency of the device interrupt handler. A further requirement is the ability to send asynchronous data streams to a group of tasks without waiting for a reply.

8.4.2 Interrupt Handling.

Interrupts need to be serviced with as little run-time overhead as possible. Ada9X needs to supply an interrupt facility with low and predictable run-time overhead and allow interrupts to be processed at an appropriate priority. The issues involved here are to permit the interrupt handling code to mask the interrupt that caused the handler to be activated, handle interrupts with minimal scheduler overhead when data must be passed to tasks of a lower priority, and permit the handling of nested interrupts where this is facilitated by the hardware architecture. Furthermore, there is a requirement to allow an interrupt binding to be changed dynamically to recover from a hardware fault or as a result of a change in the system status.

8.4.3 Distributed Systems.

Distributed systems are expected to proliferate in future applications as has been outlined in this thesis. Typically distributed systems will have to permit dynamic configuration to optimise the use of resources. Fault tolerance must be addressed since frequently the object of a multiple processor system is to allow continued application processing when a part of the system fails. A distributed system may consist of a number of different processors under different operating systems. In this case the current Ada standard may not be applicable. Ada9X is expected to specify an approach to extend the language minimally to accommodate the logical and physical separation of the application. The Ada code must be logically structured so as to execute over a number of separate computers. This requires the ability to partition the code to run on

multiple processors while maintaining the same degree of type checking that exists in a single computer system. The requirements group suggested that the following enhancements may facilitate distribution:

1) *Specification of the exact semantics (behaviour and timing) of entry and remote subprogram calls, and exception propagation.*
2) *The specification of semantics to deal with hardware failure and recovery.*
3) *Treatment of independent clocks with different timing precisions.*

Further, there is a requirement for dynamic configuration to allow the allocation of code to a number of processors upon which the application executes. The fundamental model of Ada83 is to bind a single large program and elaborate its library packages and this is somewhat contrary to the requirement for dynamic reconfiguration. A minimal solution would be to rephrase certain of Ada's rules to allow for the kinds of reconfiguration that are being practised and have been discussed in this thesis. It has been stressed that the proposals are draft considerations that require much refinement. As such it is unlikely that the final implementation of the language will reflect the requirements that are proposed above.

This chapter has considered some of the factors that would affect the performance of an application making use of the communications environment. A few of the limitations of Ada83 have been raised and these will have to be addressed by Ada9X to satisfy the current market. Ada9X has not been fully addressed as that would be far beyond the scope of this thesis.

CHAPTER NINE

Conclusion

This thesis has been concerned with the use of Ada in distributed embedded real-time control systems for process control applications that are based upon low cost personal computer local area networks.

A mechanism was designed in order to allow applications operating at nodes on the LAN to communicate for successful process control. A communications package made use of NetBIOS, as the underlying protocol, to allow for reliable internode communication. Two sample applications were implemented to show that the communication package functions as intended.

The sample applications showed that the communications package reliably transferred messages so that small processes could be controlled. The performance of the communications system is closely related to that of the underlying communications protocol, NetBIOS, and the local area network. The ARCNET local area network that was used in this project uses a token bus protocol whose data rate is 2Mbs. This transfer rate was found to be adequate for the sample applications. For applications that require higher transfer rates, the ARCNET could be replaced by any other local area network that supports a NetBIOS emulator.

The reliability of the communications system is heavily dependent upon the NetBIOS protocol. Where NetBIOS provides a successful mechanism to transport the information from one node to another the purpose of the communications package is to store the received message in a buffer. The efficiency and reliability of the communication system is then dependent upon sufficient space being available in the receive buffer. These factors can be controlled by the application programmer. The NetBIOS emulator, the message size and the buffer size can be chosen by the application programmer and tuned to suit the application. Novell provides several fields within its *shell.cfg* file to fine tune the NetBIOS emulator (these fields are discussed in Appendix B).

The performance and reliability of the system would have been enhanced by designing a network communications protocol, similar to the Manufacturing Applications Protocol, MAP,

specifically for process control. One of the objectives of this project was to make use of a current, widely available protocol for communications. NetBIOS best fulfilled the requirements of those protocols that were readily available.

The reconfiguration facilities of the system permit the application programmer to alter the logical structure of the communication network. It was designed to allow an application programmer to introduce a new application into the control loop. This facility may also be used to replace a node already in existence to allow the old application to be updated and corrected. The reconfiguration facility was shown to operate effectively in the pump control system.

The sample applications have shown that the communications package is easy to use. Once the application programmer has identified the partners with which a particular node needs to communicate, the programmer creates a configuration file. The application specifies at which point initialisation occurs and the routine constructs the required network links. The initialisation routine effectively reduces the effort required to initialise the adapter and create the network links. It minimises the initialisation effort that is common to all connection oriented interfaces. The application can then transfer and receive information from any named node. The package also provides for debugging calls to facilitate the programming of the communications system.

With vast changes expected when Ada9X is released in its final version, the value of this type of approach towards distributed Ada would need to be re-evaluated. Further study would be required to fully assess all the available communication protocols, perhaps with a view of introducing a new model. Certainly the applications for distributed systems are increasing and the lessons learnt from an exercise such as this are widely applicable.

APPENDICES

Appendix A

This appendix contains all the command codes for a NetBIOS adapter as released by IBM corporation [29].

NETBIOS COMMAND CODES

NetBIOS adapter interrupt vector

NETBIOS_INTERRUPT_5C . . . . . : constant := 16#5C#;

NetBIOS general commands

NCB_RESET_ADAPTER_COMMAND_WAIT_ONLY . . . . . : constant := 16#32#;
NCB_CANCEL_CMD_WAIT . . . . . : constant := 16#35#;

NetBIOS name support

NCB_ADD_A_NAME_WAIT . . . . . : constant := 16#30#;
NCB_ADD_A_NAME_NO_WAIT . . . . . : constant := 16#B0#;
NCB_DELETE_A_NAME_WAIT . . . . . : constant := 16#31#;
NCB_DELETE_A_NAME_NO_WAIT . . . . . : constant := 16#B1#;

NetBIOS session support

NETBIOS_CALL_WAIT . . . . . : constant := 16#10#;
NETBIOS_CALL_NO_WAIT . . . . . : constant := 16#90#;
NETBIOS_HANG_UP_WAIT . . . . . : constant := 16#12#;
NETBIOS_HANG_UP_NO_WAIT . . . . . : constant := 16#92#;
NETBIOS_LISTEN_WAIT . . . . . : constant := 16#11#;
NETBIOS_LISTEN_NO_WAIT . . . . . : constant := 16#91#;

NetBIOS session transfer commands

NETBIOS_SEND_MESSAGE_WAIT . . . . . : constant := 16#14#;
NETBIOS_SEND_MESSAGE_NO_WAIT . . . . . : constant := 16#94#;
NETBIOS_RECEIVE_MESSAGE_WAIT . . . . . : constant := 16#15#;

```
NETBIOS_RECEIVE_MESSAGE_NO_WAIT  . . . . . : constant := 16#95#;

NETBIOS_RECEIVE_ANY_MESSAGE_WAIT . . . . . : constant := 16#16#;
NETBIOS_RECEIVE_ANY_MESSAGE_NO_WAIT  . . . . . : constant := 16#96#;
```

NetBIOS datagram support

```
NETBIOS_RECEIVE_BROADCAST_DATAGRAM_WAIT . . . : constant := 16#23#;
NETBIOS_RECEIVE_BROADCAST_DATAGRAM_NO_WAIT  : constant := 16#A3#;
NETBIOS_SEND_BROADCAST_DATAGRAM_WAIT  . . . . . : constant := 16#22#;
NETBIOS_SEND_BROADCAST_DATAGRAM_NO_WAIT . . . : constant := 16#A2#;
```

Appendix B

B.1 NETBIOS ERROR CODES

	Return Code Name	*Return Code Meaning*	*Recommended Actions*
00	Successful completion	Command completed successfully.	No action required
01	Invalid Buffer Length Send Datagram, Broadcast	Illegal buffer length	Specify the correct size for the buffer length and try again.
03	Invalid Command	The command code used was incorrect.	Reissue the command with the correct code.
05	Command timed out	For a call the system time out has elapsed, otherwise the time out specified in the call or listen has elapsed	For a call try again later otherwise the session has terminated abnormally
06	Message incomplete	Part of the message received because the receive buffer is too small.	Issue another receive for session commands in other cases the remainder of the data is lost
08	Illegal local session number	The session number specified is not valid	Specify an active session
09	No resource available	Not enough space available in the adapter to issue the command	Reissue the command later
0A	Session closed	The session has been closed either locally or remotely	No action required
0B	Command cancelled	Notification received that the command is cancelled	No action is required
0D	Duplicate name in local name table	Attempt to specify a name that already exists in name table	Specify another name
0E	Name table full	Up to 16 names have aleady been added	Issue a delete name so an entry will become available

0F	Command complete, has active sessions but now de-registered	The name to be deleted is currently active but is de-registered prior to deletion	Close all sessions using this name so the delete name can complete
11	Local session table full	No available entries in the session table	Wait until a session is closed or increase the sessions available when the adapter is reset
12	Session open rejected	No listen command is outstanding on the remote computer	Wait until a listen is issued on the remote computer
13	Illegal name number	Invaild name number	You must use the original name number that was assigned to the name
14	Cannot find name called or no answer	The called name specified cannot be found or no answer	Verify that the called nam is correct or reissue if the remote computer is busy
15	Name not found	The name was not in the name table or a * or 00h was in the first field of the name	Retry with another name and verify that it is correct
16	Name in use on remote adapter	Unique names may only be used once an the network	Specify another name
17	Name deleted	Name has been deleted and there are no outstanding commands for that name	No action required
18	Session ended abnormally	The remote computer was powered off, or a send timed out, or a link broken	Check the remote end, or reestablish the session
19	Name conflict detected	Network protocol has detected twoor more identical names on the network	Everyone on the network should delete that name immediatly
1A	Incompatible remote device	Unexpected protocol packet received	Verify that all units agree on the network protocols

21	Interface busy	You called the BIOS out of an interrupt handler routine in process	Return from the interrupt handler and try again later
22	Too many commands outstanding	The maximum number of commands are outstanding	If not at the maximum number refer to the reset command else try again later
23	Invalid number in LANA field	You tried to specify a number other than 00 or 01	Correct the number and try again
24	Command completed while cancel occurring	Attempt to cancel a command that had already completed	No action required
25	Command not valid to cancel	An invalid attempt was made to cancel a command	See cancel command to see what commands may be cancelled
4X	Unusual network condition. X may be any hex value	The BIOS has detected an unusual condition in the network	Either retry or reset the command
50-FE	Adapter malfunction	The adapter has detected an internal problem	Retry the operation
FF	Command pending status	The command is still pending	No action required

B.2 ADDITIONAL ERRORS FROM THE COMMUNICATIONS PACKAGE

300	Insufficient nodes		
301	Sending name not found	The destination name does not exist in the network management list	Either add a session to that name or chose a valid name
302	Message not received	The message requested was not received	See that is a valid receiver routine at the remote node
303	Cannot reset the adapter	All communication channels must be closed before the adapter is reset	Close the remaining open communication channels with a hang up command

304	Cannot initiate channel	Cannot initiate a channel with ones own host node name	Initiate a channel with a valid name
305	Connection already exists	Attempt to initiate a channel with a connection that already exists	No action required
306	All connections in configuration file not made	All the requested communication channels are not completed	Either attempt to initiate a channel here later or do not communicate with this node
307	Configuration file connections have been made	All configuration connectiosn have already been made	No action required
308	The redirection attempt failed	The attempt at redirection failed	Affirm that the connections that were to be redirected are valid

THE SHELL CONFIGURATION FIELDS

The shell configuration fields are used by Novell to manipulate the workstatons network environment or configuration. The shell configuration file is created with a DOS text editor. This section will deal with those fields that affect the operation of the Novell NetBIOS emulator and not those fields that are concerned with IPX/SPX. The reader is referred to reference [37] for further information.

NetBIOS Abort Time Out

This parameter adjusts the amount of time, in ticks, that NetBIOS will wait without receiving any response from the other side of the session.

Default 540.

NetBIOS Broadcast Count

This parameter, when multiplied by the NetBIOS Broadcast Delay, determines the total time it takes to broadcast a name across the network. This value should be increased if their are many LAN segments that require NetBIOS support.

NetBIOS Broadcast Delay

This parameter, multiplied by the NetBIOS Broadcast Count number, determines the total time (in ticks) it takes to broadcast a name across the network and reflects the amount of traffic on the network.

Default when NetBIOS Internet = on : 36
Default when NetBIOS Internet = off: 18

NetBIOS Commands

This sets the number of commands that may be pended at once. The default of 12 is usually sufficient. If NetBIOS command error 22 occurs the parameter must be increased.

Default 12.

NetBIOS Internet

If you applications are being run on a single network with a dedicated file server this parameter will speed up delivery of packets if the value is set to off.

NetBIOS Listen Timeout

This parameter adjusts the time that NetBIOS will wait (when no packets are received from the other side of a session), before it requests a "keep-alive" packet from the other side to assure the session is still valid.

Default: 108 ticks (about 6 seconds)

NetBIOS Receive Buffers

This parameter configures the number of IPX receive buffers that NetBIOS uses. Increase this value where incoming burst traffic situations are common.

Default: 6

NetBIOS Retry Count

The retry count determines the number of times NetBIOS will resend a packet to establish a session with a remote partner.

Default when NetBIOS Internet = on : 20
Default when NetBIOS Internet = off: 10

NetBIOS Retry Delay

This parameter affects NetBIOS session commands only. Retry delay sets the delay (in ticks) between each packet that NetBIOS sends during an attempt to establish a session.

Default: 10 ticks (about .5 seconds)

NetBIOS Send Buffers

This parameter configures the number of IPX send buffers that NetBIOS uses. This value should be increased where incoming burst traffic situations are common.

Default: 6 buffers

NetBIOS Sessions

This parameter configures the maximum number of virtual circuits that NetBIOS can support at the same time. Specify any value from 4 to 250.

Default: 32 sessions

NetBIOS Verify Timeout

This parameter adjusts the frequency at which NetBIOS sends a "Keep alive" packet to the other side of a session to preserve the session.

Default: 54 ticks (about 3 seconds)

Npatch

This parameter patches any location in the NETBIOS.EXE data segment with any value.

Default: not used unless specified

Appendix C

THE NETBIOS PACKAGE SPECIFICATION

```
with errors,System,definitions;
use  errors,System,definitions;

package NETBIOS is

------------------------------------------------------------------------------
----------------------        NetBIOS Commands        ----------------------
------------------------------------------------------------------------------

netbios_interrupt_5c                                  : constant := 16#5C#;
ncb_reset_adapter_command_wait_only                   : constant := 16#32#;
ncb_add_a_name_wait                                   : constant := 16#30#;
ncb_add_a_name_no_wait                                : constant := 16#b0#;
ncb_delete_a_name_wait                                : constant := 16#31#;
ncb_delete_a_name_no_wait                             : constant := 16#b1#;
ncb_cancel_cmd_wait                                   : constant := 16#35#;

netbios_call_wait                                     : constant := 16#10#;
netbios_call_no_wait                                  : constant := 16#90#;
netbios_hang_up_wait                                  : constant := 16#12#;
netbios_hang_up_no_wait                               : constant := 16#92#;
netbios_listen_wait                                   : constant := 16#11#;
netbios_listen_no_wait                                : constant := 16#91#;

netbios_receive_broadcast_datagram_wait               : constant := 16#23#;
netbios_receive_broadcast_datagram_no_wait            : constant := 16#a3#;
netbios_send_broadcast_datagram_wait                  : constant := 16#22#;
netbios_send_broadcast_datagram_no_wait               : constant := 16#a2#;

netbios_send_message_wait                             : constant := 16#14#;
netbios_send_message_no_wait                          : constant := 16#94#;
netbios_receive_message_wait                          : constant := 16#15#;
netbios_receive_message_no_wait                       : constant := 16#95#;

netbios_receive_any_message_wait                      : constant := 16#16#;
netbios_receive_any_message_no_wait                   : constant := 16#96#;


------------------------------------------------------------------------------
----------------------          ERROR CODES           ----------------------
------------------------------------------------------------------------------

type error_array is array (0..40) of string(1..52);
net_errors : constant error_array := (
```

```
    0 => "00 Successful completion                                  ",
    1 => "01 Invalid Buffer Length                                  ",
    2 => "Command Not Documented                                    ",
    3 => "03 Invalid Command                                        ",
    4 => "Command Not Documented                                    ",
    5 => "05 Command Timed Out                                      ",
    6 => "06 Incomplete Received Message                            ",
    7 => "07 Local No-Ack Command Failed                            ",
    8 => "08 Invalid Local Session Number                           ",
    9 => "09 No Resource Available                                  ",
   10 => "10 Session Has Been Closed                                ",
   11 => "11 Command Was Cancelled                                  ",
   12 => "Command Not Documented                                    ",
   13 => "13 Duplicate Name In Netbios Table                        ",
   14 => "14 Netbios Name Table Full                                ",
   15 => "15 Name Has Active Sessions and is Now Deregistered       ",
   16 => "Command Not Documented                                    ",
   17 => "17 Netbios Local Session Table Full                       ",
   18 => "18 Session Open Rejected - no pending Listen              ",
   19 => "19 Illegal name Number                                    ",
   20 => "20 Cannot Find Name Called Or No Answer                   ",
   21 => "21 Refer to Reference (15H)                               ",
   22 => "22 Name In Use On Remote Adapter                          ",
   23 => "23 Name Deleted                                           ",
   24 => "24 Session Ended Abnormally                               ",
   25 => "25 Name Conflict Detected                                 ",
   26 => "26 Incompatible Remote Device (PC Network)                ",
27..32 => "Command Not Documented                                   ",
   33 => "33 Interface Busy                                         ",
   34 => "34 Too Many Commands Outstanding                          ",
   35 => "35 Invalid Number in NcbLanaNum Field                     ",
   36 => "36 Command Completed While Cancel Occuring                ",
   37 => "37 Command Not Documented                                 ",
   38 => "38 Command Not Valid To Cancel                            ",
39..40 => "Command Not Documented                                   ");

mask_low_word      : constant := 16#FFFF#;
max_adapter_number : constant := 1;
max_session_count  : constant := 254;
max_command_count  : constant := 255;
max_names          : constant := 254;

--- type address defined in system

NCB_LOCATION               : Address;
```

```
-------------------------------------------------------------------------------
-------------      GENERAL NETBIOS FUNCTIONS      -------------------
-------------------------------------------------------------------------------

function check_for_netbios return boolean;

procedure netbios_adapter_reset(nb_adapter_number    : unsigned_byte;
                                nb_session_count     : unsigned_byte;
                                nb_command_count     : unsigned_byte;
                                ncb_pointer          : ncb_access);

procedure add_netbios_name(new_adapter_name  : netbios_adapter_names;
                           wait              : boolean;
                           int_address       : address;
                           NCB               : in out NCB_ACCESS);

-------------------------------------------------------------------------------
-------------------  NETBIOS LISTEN & CALL ROUTINES  ----------------------
-------------------------------------------------------------------------------

procedure netbios_listen_routine(destination_name :  netbios_adapter_names;
                                 ncb_name         :  netbios_adapter_names;
                                 int_address      :  system.address;
                                 ncb_sto          :  unsigned_byte;
                                 ncb_rto          :  unsigned_byte;
                                 wait             :  boolean;
                                 ncb              :  in out ncb_access );

procedure netbios_call_routine(destination_name  :  netbios_adapter_names;
                               ncb_name          :  netbios_adapter_names;
                               ncb_sto           :  unsigned_byte;
                               ncb_rto           :  unsigned_byte;
                               wait              :  boolean;
                               ncb               :  in out ncb_access );


-------------------------------------------------------------------------------
---------------  NETBIOS SEND AND RECEIVE ROUTINES  ---------------
-------------------------------------------------------------------------------


procedure  netbios_send_message_routine(message : message_type;
                                        lsn     : in unsigned_byte;
                                        wait    : in boolean;
                                        int_address : in system.address;
                                        ncb     : in out ncb_access);
```

```
-------------------------------------------------------------------------------
---------- NETBIOS TERMINATION ROUTINES ----------
-------------------------------------------------------------------------------

  procedure netbios_hang_up(ncb  : in out NCB_ACCESS;
                            lsn  : in unsigned_byte;
                            wait : boolean);

end NETBIOS;
```

Appendix D

D.1 THE NETBIOS CONTROL BLOCK

The NetBIOS package that is written in Ada and is used by the communications package issues NetBIOS commands by constructing a netbios control block of sixty-four bytes [29]. The NetBIOS application (referred to as application in this appendix) completes various fields that differ according to the command that is issued. Failure to complete the fields correctly can cause the user's machine to hang. After the control block has been completed the es:bx register is pointed at the first memory location of the block and an INT 5Ch interrupt request is generated.

Offset	Field Identifier	Length
00	Command	01
01	Return Code	01
02	Local Session Number	01
03	Name Number	02
04	Buffer Address	04
08	Buffer Length	01
10	Call Name	16
26	Name (local)	16
42	Receive Time Out	01
43	Send Time Out	01
44	Post Routine Address	04
48	LANA Number	01
49	Command Complete Flag	01
50	Reserved Field	14

Figure D.1 The Netbios Control Block Structure
Taken From Schwaderer [30]

NCB Command Code. The NCB command field is a one byte field specifying the NetBIOS command. If the high order bit of NCB command is binary zero then the wait command option is specified, otherwise the no-wait option is set. The Cancel, Reset and Unlink commands posses only the wait option. Selecting the wait option requests NetBIOS to return control when the adapter completes the command. When the command completes, the AL register and the NetBIOS return code field contain the completion code. Selecting the no-wait option requests that NetBIOS return control to the application while the command is pending completion. A post routine (equivalent to an interrupt handler) is given control when the command completes, if the post routine address is not zero.

NCB Return Code. The NetBIOS return code field is a one byte field that contains the commands return code. A non-zero value indicates an error, and the error codes are listed in Appendix B.

NCB Local Session Number. The local session number is a one byte field containing the local session number associated with a command. NetBIOS assigns the value in an incremental, modulo 255, round robin manner.

NCB Name Number. The NetBIOS name number is a one byte field containing the name table number and is issued in much the same manner as the local session number.

NCB Buffer Address. The buffer address is a four byte field containing a memory pointer to a data buffer in the OFFSET:SEGMENT format (IBM PC).

NCB Buffer Length. This is a two byte field indicating the size of the buffer pointed at by the buffer address field.

NCB Call (remote) Name. This is typically a sixteen byte field containing a remote name associated with a request. All sixteen bytes are significant and are used.

NCB (Local) Name. The NetBIOS local name is a sixteen byte name associated with a request. The first character may not have a binary zero or an asterisk.

NCB Receive Time Out and Send Time Out. These are one byte fields used with the call and listen commands to specify any number of half second periods that a command may wait before timing out. The time out threshold is established at session creation and may not be altered subsequently. Specifying a zero time out indicates that there is no time out associated with this command.

NCB Post Routine Address. The post routine address is a four-byte field containing a memory pointer to a routine that is executed when the command completes. NetBIOS inspects this option when the command has specified the no-wait option.

NCB LANA Number. This is a one byte field that indicates which adapter should handle the command. The primary adapter is zero.

NCB Command Complete Flag. This is a one byte field that indicates the status of the command that has been issued in the no-wait format. The value here is FFh until the command completes with the correct return code.

NCB Reserved. This is a fourteen byte field that NetBIOS may use to return extended error information. NetBIOS also uses it as a work area when processing the request. If this area is tampered with the behaviour of NetBIOS may become unpredictable.

Appendix E

The Communications Package Users Manual

E.0 INTRODUCTION

The communications package provides users with a mechanism to transfer messages across a local area network in order to perform process control applications. Issues relating to the performance of the communications environment, such as the throughput of the local area network, the underlying protocol and the real-time scheduler have been discussed previously. This chapter provides a step by step instruction on the use of the communication interface.

E.1 USING THE COMMUNICATIONS SYSTEM

E.1.1 System Requirements

The communications package was written and compiled using Meridian Ada 4.1.1 for the personal computer. When designing and implementing a controller using the communications package and Meridian Ada, the Meridian compiler must be operated in extended mode with sufficient memory to compile realistically sized packages.

Some functions offered by the communications system have been written in 8086 assembly language to improve the package's speed and efficiency. These routines have been included in the communications library using Meridian's assembly language interface.

The communications system is based upon an underlying NetBIOS protocol and this must be present, in the form of a NetBIOS emulator or ROM, before the communication system will operate satisfactorily.

E.1.2 System Configuration

Before any compilation is attempted the user must execute the install.bat file on the communication system diskette. This will copy the communication system files into the path

specified by the application programmer. The programmer must then direct the Ada compiler to search the communications library in addition to the default library. If the location of the application is c:\app1 then the required commands are:

```
c:\app1 > a:install c:\comms
c:\app1 > lnlib c:\comms\ada.lib
```

Before attempting to use the communication system the application programmer must identify the number of nodes and the connections that are required between those nodes for the distributed control system. Once the number of nodes and the interconnections between the

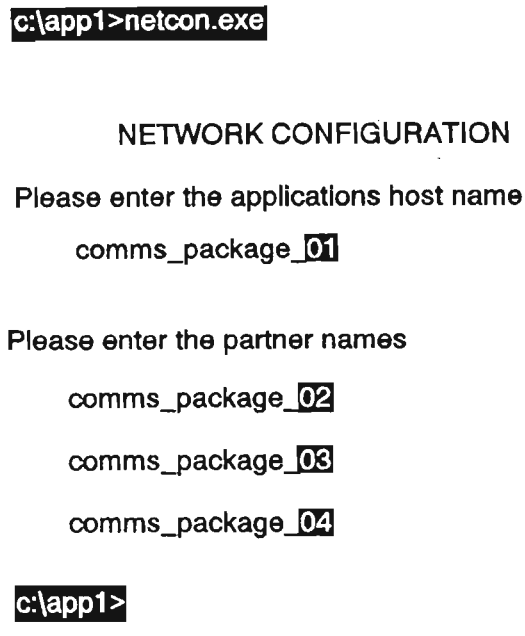

Figure E.1 The Netcon Utility

nodes have been determined the application programmer must use the utility, *netcon.exe* (provided with the communications system), to create configuration files for each application on the network (refer to figure E.1). The application programmer must enter the name of the application at each node and the partners with which it will communicate. The configuration file, called *netcon.fig*, must reside in the directory from which the application will execute. The names of the applications all begin with "comms_package_" and the application programmer must add the final two characters to complete the name. Each application name must be unique.

It is not essential that partners are listed in the configuration file since it is possible for the application to instantiate a communication link explicitly, at a later stage. The system was designed in this manner to reduce the programming effort when constructing the logical network.

E.1.3 Initialising the communications package

The communications package has been designed as a generic library package with two integer input parameters, the message and buffer sizes. The application programmer must determine what maximum message and buffer sizes are required for the application so the communications package can be instantiated with the relevant information. This permits the communication routine to set aside a guaranteed storage area for message receipt. The package *Relay*, that is shown in the example code throughout this manual, is the communications driver for the Hot-Line communications system (refer to code E.1). The complete package is documented in section E.3; all code for the Hot-Line communication system is included on the demonstration diskette. The instantiation of the communications package (communications(message_length, buffer_length)) is shown below:

```
with COMMUNICATIONS;
package comms is new communications(64,3);
with COMMS;

package body RELAY is
        . . . . .
        . . . . .
end RELAY;
```

CODE E.1

E.1.4 Starting the communications system

The communications package can operate on networks that make use of NetBIOS emulators rather than NetBIOS ROMs. To determine whether NetBIOS exists at the node, in the form of a ROM or emulator, a *check_for_netbios* call is provided. The test is based upon those used for the IBM PC-XT and PC-AT computers. This test is not foolproof, although it offers more

protection than would otherwise exist. The original IBM BIOS POST procedures initialise interrupts that are not required to OFFSET:SEGMENT values of 0000:0000. Issuing a NetBIOS request to one of these machines, when the interrupt has not been initialised, is highly likely to hang the machine. The PC-AT BIOS points uninitialised vectors to an *iret* instruction. Thus a request to one of these machines is harmless though unproductive.

```
-------------------------------------------------------------------------
--        Perform the communication system initialisation              --
-------------------------------------------------------------------------
procedure init(failure_code : out integer) is
        code : integer := 0;
        name : string(1..16);
begin
-- does netbios exist
        if comms.check_for_netbios then
-- perform the communication system setup routine
                comms.system_setup(failure_code => code,
                                        retry_count => 5);
                failure_code := code;
        else
                tty.put_line(" NetBIOS was not installed. ");
        end if;
end init;

end RELAY;
```

CODE E.2

Once the application has determined that NetBIOS is present, it must initialise the communications system through the *system_setup* routine before any communication operations are performed (code E.2). The *system_setup* call resets the adapter, adds the NetBIOS node name and initiates communication channels with the partners that are specified in the configuration file. The application must specify a *retry_count*, which is the maximum number of times the communications package will attempt to initialise a channel with any partner. A failure code is returned upon completion, indicating the status of the attempt.

If the failure code returned is zero the initialisation was successful and the application may send or receive messages from the partners listed in the *netcon.fig*. If the initiation of a channel with a particular partner failed, the name of the partner may be obtained through a

configuration_name call (code E.3). At any stage after the initialisation process has completed, the application may attempt to initiate a channel with this partner using the *initiate* call. If a channel already exists or if the partner does not exist on the network, this is indicated through a failure code and channel initiation is not permitted. A complete list of error codes is contained in Appendix B.

```
procedure init(failure_code : out integer) is
        code : integer := 0;
        name : string(1..16);
begin
-- does netbios exist
        if comms.check_for_netbios then
-- perform the communication system setup routine
            comms.system_setup(failure_code => code,
                               retry_count => 5);
            failure_code := code;
            if code = 0 then
-- while all partners have not been connected retry the connection routine
                while new_code /= 107 loop
-- get the name of the partner that could not be found
                    comms.configuration_name_call(name => name,
                                                  failure_code => new_code);
-- if such a partner exists then initiate a channel with it
                    if new_code = 106 then
                        comms.initiate(name => name,
                                       failure_code => code);
                    end if;
                end loop;
-- otherwise the communication channel initiation failed
            else
                tty.put_line(" Communications Initiation Failed ");
                tty.put_line(" Failure Code : " & integer'image(code));
            end if;
        else
            tty.put_line(" NetBIOS was not installed. ");
        end if;
end init;
```

CODE E.3

E.1.5 Sending and receiving messages

Once channels have been created the application may send messages to a destination node or retrieve messages from the receive buffer. The receive buffer stores all messages sent by other applications to the node. It is managed by the communications package.

The application may send messages to remote applications in a synchronous or asynchronous mode. The asynchronous send promotes concurrency through returning control to the application as soon as the command is pended. The command is likely to complete while the application is busy with another task so the communications interface provides a function, *fetch_send_failure_code,* which allows the application to obtain the failure code of the last asynchronous send pended. Only the failure code of the last asynchronous send is retained since other send alternatives are intended for use when full acknowledgement is required. Maintaining the failure codes for a list of asynchronous sends and matching them to the corresponding send request represents significant overhead.

```
task body transporter is
        Message : Message_Type(1..message_length):=(1..message_length => ' ');
begin
-- initialisation entry
        accept go;
        SEND_MESSAGE_LOOP:
        loop
            select
-- accept a message to send
            accept send_message(message : in message_type;
                                code : out integer) do
-- display the partner name that the message is sent to
                device_display.display_host(host_name);
-- send (synchronous) the message with acknowledge and the message may not be
-- overwritten before it is read
                comms.send(node_name => partner_name,
                           message => message,
                           discard_msg => 1,
                           ack => 0,
                           failure_code => code);
            end send_message;
            or
                terminate;
            end select;
        end loop SEND_MESSAGE_LOOP;
end transporter;
```

Code E.4

The synchronous send routine provides several choices that reflect the importance of the message to be sent (code E.4). In cases when it is essential that the target application obtains and processes a message, the communications interface may specify that when the message is

Send Alternatives	Acknowledge (Integer)	Discard Message (Integer)
No Buffer Ack/ Cannot Overwrite Until Read	0	1
No Buffer Ack/ Can Overwrite Before Read	0	2
Buffer Ack/ Cannot Overwrite Until Read	1	1
Buffer Ack/ Can Overwrite Before Read	1	2
Send Emergency Directive	3	Target Interrupt No.
Asynchronous Send/ Cannot Overwrite	5	1
Asynchronous Send/ Can Overwrite	5	2

Figure E.2 The Send Alternatives

received and space is available within the message buffer, a positive acknowledgement is despatched to the sending node. The message will only be overwritten once it has been processed by the application. If no buffer space is available the sending node is informed accordingly. If messages updating the information are sent at a faster rate than the messages are processed by the target application, the *discard_msg* field may be set equal to two, so new messages will overwrite the old, unread messages to ensure that the application processes the latest information.

The *send with buffer acknowledge* will check that a message can be transported successfully, with the receiver sending back an acknowledgement message that buffer space is available. If this option is too expensive, due to the acknowledgement message, the user may use the *send with no buffer acknowledge*. This alternative offers the application the assurance that the physical communication link is available and if there is no space in the buffer, it is because the application still has a number of messages to process. NetBIOS confirms that state of the communications link and the communications package returns standard NetBIOS error codes.

```
-------------------------------------------------------------------------
--    Retrieves the latest message from the comms buffer     --
-------------------------------------------------------------------------
task body receive is
        last: boolean :=true;
        message_pointer,code : integer :=0;
        sending_name : string(1..16);
        message:message_type(1..message_length):=(1..message_length => ' ');
begin
-- the initialisation entry
        accept go;
        loop
-- retrieve the message from the receive buffer
            comms.get_latest_message(message => message,
                                     msg_pntr => message_pointer,
                                     name => sending_name,
                                     failure_code => code);
-- if the message is successfully retrieved then proceed
            if code = 0 then
-- display the partner that sent the message
                device_display.display_remote(sending_name);
-- display the message itself
                device_display.Display_Message(message,last);
            end if;
            delay 0.0;
        end loop;
end receive;
```

Code E.5

The application receives messages sent to it by requesting the communications package to read the message buffer through the *get_latest_message* function (code E.5). The communications package returns a message, the name of the node that despatched the message and the message pointer which describes the position of that message within the message buffer. The communications routine sets a flag to indicate that the message has been read. The receiver routine checks the *discard_flag* of the oldest message in the circular buffer before replacing it with a new message. If the *discard_flag* is zero (the message has been read) or two (the message may be overwritten if desired - the *discard_msg* field is one) then the new message is dumped in the message buffer. The new message is discarded if the *discard_flag* for the last message is one since this implies that there is no space in the message buffer. This indicates that the message has not been read and cannot be overwritten (the *discard_msg* field of the receive record on the destination is one). When all messages have been read the *get_latest_message* function returns a failure code to specify that no unread messages exist.

For emergencies the communications package provides an application with hooks in the form of interrupt vectors that may be allocated. To make use of a hook the application program must set the *ack* and *discard_msg* fields of the send command to three and to the interrupt number of the hook respectively. It is crucial that the destination has a corresponding interrupt entry for the number; the application may use any interrupt number that is not in use by the operating or communication system. A message is despatched to the destination with information pertaining to the emergency. The receiver routine at the target destination intercepts that message and issues a software interrupt for that particular hook. It is then the responsibility of the application programmer to ensure that the corresponding handler is available. These are alarm handlers and suspend the operation of the rest of the system for the duration of the entry call. Thus they must be used with due caution and if possible, the execution of the handler must be completed outside of the scope of the entry call. The entry call is just a signal to a user defined alarm handler and any information that was passed in the signal message must be explicitly received by the handler without delay from the *receive_alarm_message* routine. The limitations of Meridian Ada do not permit the passing of any information through an interrupt entry call as an in parameter. This has been specified as an implementation option in the Ada Language Reference Manual [18].

The above approach caters for the circumstance where it is absolutely necessary to request a message and obtain it as soon as possible, from a particular node with the most current information available through using the alarm handler hooks that are provided. The destination node must have a handler with an interrupt entry that corresponds to a particular handler number, written by the application programmer, which will create a message and despatch it to the requesting node. An application sends an alarm signal message to the above destination specifying an interrupt entry number. The receiver routine, at the destination, intercepts the message and issues the appropriate software interrupt. Control branches to the Ada task defined by the application programmer to deal with just such an emergency which creates the message and despatches it to the requesting node.

E.1.6 The Termination Process

Once communication has completed the application must close the communication link. The communication package offers the facility to close a channel if the communication with that particular node is no longer necessary. One of the applications issues a *hang_up* command, passing the name of the partner whose session entry is to be closed. The channel is closed and if communication is attempted with this node a failure code is returned indicating that the session no longer exists. When all sessions have been closed the application may issue a *reset* command which removes all trace of the node on the network. Once the *reset* has been issued the node has undergone its full lifecycle.

```
-------------------------------------------------------------------------
--          Close the communication system down                        --
-------------------------------------------------------------------------
procedure finish_communications is
        code : integer := 0;
begin
-- perform the communications hangup with the remote partner
        comms.hangup(name => partner_name,
                     code => code);
-- if successful then the connection is closed
         if code = 0 then
            tty.put_line(" The Connection Has Been Closed ");
         end if;
-- terminate the transporter task, receive task
        transporter.terminate_transporter;
        abort receive;
-- perform the communications reset to reset the comms harness
        comms.reset(failure_code => code);
        if  code = 0 then
            tty.put_line(" Adapter is reset ");
        end if;
        tty.put_line(" Failure code is : " & integer'image(code));
end;
```

Code E.6

E.1.7 Supplementary Calls

The communication interface provides other function calls that are of importance. The redirection facility has the most effect upon application processes executing over the network.

Consider, in figure E.3a, two nodes A and B that communicate with one another and that represent a distributed process control system. In a given situation, a further node may represent a function that is not currently implemented in the industrial process. This new application function may initialise itself on the network in the conventional way, as discussed above. It may issue a *redirect* command, passing as parameters the destination node, Node A, and the node with which it communicates, Node B. This function sends an instruction to the communications package of node A to replace the session number that exists for the communication channel AB with the session number of the channel (with node C) that sent the redirection directive. The application at node A still believes it despatches data to node B although it in fact communicates with C.

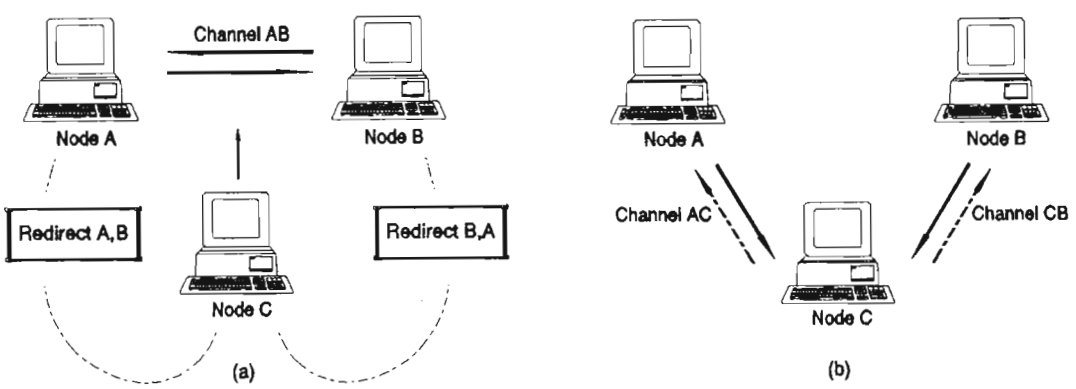

Figure E.3 The Redirection Operation

If a similar instruction is directed to node B, node B will despatch its data to C. It is node C's responsibility to route the information that A and B require to continue operating successfully and used to obtain directly from one another (this is illustrated by the dotted, arrowed lines in figure E.3b). When an insertion is achieved in the above manner the session that existed between A and B remains and must be removed explicitly through a *hang_up* command. If the channel is left in place it is not visible to either application at node A or B and consequently cannot be used. If the channel had not been removed and A had been redirected as discussed above and then A issued a *hang_up* command, the session number that is associated with channel AC would be replaced with the original session number that existed for the channel with node B, channel AB, and so allow the system to return to its original configuration. The redirection facility allows for a new application node to be developed and tailored independently

of the actual process executing over the network and inserted without bringing the whole process to a halt.

An additional call, function *my_name*, allows an application to verify the host node name that the communications package is using. Further, the application may issue an *obtain_connection_array* call to obtain a complete list of partners with which the application has valid sessions and may communicate.

The communication package also offers some debugging functions that the system designer may use to provide for a visual display of the latest message received and the full contents of the message buffer, listing the message, the place it occupies within the buffer and whether it has been read or not. The other call provided lists the contents of the network management list displaying the connections that the node has open with other nodes on the system. The following subsection contains the full communications interface specification package.

E.2 THE COMMUNICATIONS PACKAGE SPECIFICATION

with definitions;

```
PACKAGE COMMS IS
        netbios_status : boolean;
        subtype node_name_type is string(1..16);
        subtype byte is integer range 0..255;

--- Allows the Application to Fetch the Latest Message Despatched
procedure get_latest_message(message : out string;
                             msg_pntr : out integer;
                             name : out string;
                             failure_code : out integer);

--- Allows the Application to fetch the emergency message from the emergency buffer
procedure receive_alarm_message(message : out string;
                             name : out string;
                             failure_code : out integer);
```

```
--- The send routine permits an application to despatch a message to the name destination node
Procedure send(node_name : in string;
               message : in string;
               discard_msg : in integer :=2;
               ack : in integer :=0;
               failure_code : out integer);

-- Allows an application to obtain the failure code from an asynchronous send
function fetch_send_failure_code return integer;

--- Performs a redirection of the Output of the destination (previous target redirected) to the
host node
procedure redirect(destination : in string;
                   redirected  : in string;
                   failure_code : in out integer);

-- Provides a list of all the valid communication channels
procedure obtain_connection_array(name : out definitions.name_array;
                                  lgth : out integer;
                                  code : out integer);

-- Provides a list of all the channels that could not be instantiated in the initialisation
procedure configuration_name_call(name : out node_name_type;
                                  failure_code : out integer);

-- Displays the valid connections
procedure con_display;
-- Displays the full message buffer
procedure message_display;
-- Resets the NetBIOS adapter
procedure reset(failure_code : out integer);
-- Performs the Initialisation Procedure
procedure system_setup(failure_code : in out integer;
                       retry_count : in integer);

-- Returns the adapters host name
function my_name return string;

END COMMS;
```

E.3 THE HOT-LINE COMMUNICATIONS DRIVER

```
with COMMUNICATIONS;
package comms is new communications(64,3);
with DEFINITIONS, DEVICE_DISPLAY, TTY, COMMS;
use DEFINITIONS;
```

```
package body RELAY is

partner_name : string(1..16);
host_name    : string(1..16);
message_length : constant := 64;
retry_count    : constant := 5;
type names is array(1..1) of string(1..16);

-------------------------------------------------------------------------
--          Close the communication system down                        --
-------------------------------------------------------------------------
procedure finish_communications is
        code : integer := 0;
begin
-- perform the communications hangup with the remote partner
        comms.hangup(name => partner_name,
                     code => code);
-- if successful then the connection is closed
        if code = 0 then
            tty.put_line(" The Connection Has Been Closed ");
        end if;
-- terminate the transporter task, receive task
        transporter.terminate_transporter;
        abort receive;
-- perform the communications reset to reset the comms harness
        comms.reset(failure_code => code);
        if  code = 0 then
            tty.put_line(" Adapter is reset ");
        end if;
        tty.put_line(" Failure code is : " & integer'image(code));
end;

-------------------------------------------------------------------------
--          The transporter task manages and despatches                --
--              messages to the remote partner                         --
-------------------------------------------------------------------------
task body transporter is
        Message : Message_Type(1..message_length) := (1..message_length => ' ');
        last : boolean;
        loop_number : integer;
        input_character : character;
begin
-- initialisation entry
        accept go;
        SEND_MESSAGE_LOOP:
        loop
        select
```

```
-- to terminate the communication system once communication has completed
        accept terminate_transporter;
            abort transporter;
        or
-- accept a message to send
        accept send_message(message : in message_type;
                            code : out integer) do
-- display the partner name that the message is sent to
            device_display.display_host(host_name);
-- send the message with acknowledge and the message may not be
-- overwritten before it is read
            comms.send(node_name => partner_name,
                       message => message,
                       discard_msg => 1,
                       ack => 0,
                       failure_code => code);
        end send_message;
        or
            terminate;
        end select;
        end loop SEND_MESSAGE_LOOP;
end transporter;


-----------------------------------------------------------------
--     Retrieves the latest message from the comms buffer      --
-----------------------------------------------------------------
task body receive is
        last: boolean :=true;
        message_pointer,code : integer :=0;
        sending_name : string(1..16);
        message:definitions.message_type(1..message_length):=(1..message_length => ' ');
begin
-- the initialisation entry
        accept go;
        loop
-- retrieve the message from the receive buffer
            comms.get_latest_message(message => message,
                                     msg_pntr => message_pointer,
                                     name => sending_name,
                                     failure_code => code);
-- if the message is successfully retrieved then proceed
            if code = 0 then
-- display the partner that sent the message
                device_display.display_remote(sending_name);
-- display the message itself
                device_display.Display_Message(message,last);
            end if;
```

```
            delay 0.0;
        end loop;
end receive;


-------------------------------------------------------------------------
--        Perform the communication system and display        --
--        initialisation before the communication begins      --
-------------------------------------------------------------------------
procedure init(failure_code : out integer) is
        code : integer := 0;
        new_code : integer := 1;
        names : definitions.name_array(1..1);
        array_length : integer :=0;
        name : string(1..16);
begin
-- does netbios exist
        if comms.check_for_netbios then
-- perform the communication system setup routine
            comms.system_setup(failure_code => code,
                               retry_count => retry_count);
            failure_code := code;
            if code = 0 then
                while new_code /= 107 loop
-- while the partner has not been connected retry the connection routine
                    comms.configuration_name_call(name => name,
                                                  failure_code => new_code);
                    if new_code = 106 then
                        comms.initiate(name => name,
                                       failure_code => code);
                    end if;
                end loop;
                while new_code /=0 loop
                    comms.obtain_connection_array(name => names,
                                                  lgth => array_length,
                                                  code => new_code);
                end loop;
                partner_name := string(names(1));
-- get the applications own name
                host_name := comms.my_name;
                comms.reset(new_code);
            else
                tty.put_line(" Communications Initiation Failed ");
                tty.put_line(" Failure Code : " & integer'image(code));
                comms.reset(new_code);
            end if;
        else
            tty.put_line(" NetBIOS was not installed. ");
```

```
            comms.reset(new_code);
        end if;
end init;

end RELAY;
```

References

1) American National Standards Institute, Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A-1983.

2) Sloman M, Kramer J, "Distributed Systems and Computer Networks", Prentice-Hall(UK), 1987.

3) Ada9X Draft Mapping Rationale Document, February 1991, Office of the Under Secretary of Defense for Acquisition, Washington D.C. 20301, 1990.

4) Nielsen K, Ada in Distributed Real-Time Systems, McGraw Hill, 1990.

5) Enslow P.H., What is a 'distributed' system?, Computer, pp. 13-21, January 1978.

6) Barnes, J.G.P., Programming in ADA, Third Edition, International Computer Science Series, pp.9-10., 1989.

7) Cook, R.P., MOD - A Language for Distributed Programming, IEEE Transactions on Software Engineering, IEEE Computer Society, November 1980.

8) Volz R, Mudge T, Bussard G, Krishnan P, Translation and Execution of Distributed Ada Programs: Is it still Ada?, IEEE Transactions on Software, Special Issue on Ada, March 1989.

9) Atkinson C, Goldsack S, Communication between Ada Programs in Diadem, ACM Proceedings of the 2nd International Workshop on Real-Time Ada Issues, Ada Letters, vol VIII, no. 7, 1988.

10) Bishop J, Adams S, Pritchard D, Distributing Concurrent Ada programs by source translation, Software Parctice and Experience, volume 17, 1987.

11) Hutcheon A, Wellings A, Supporting Ada in a Distributed Environment, lications in Ada, ACM Proceedings of the 2nd International Workshop on Real-Time Ada Issues, Ada Letters, vol VIII, no. 7, 1988.

12) JHA R, Eisenhauer G, [1989], Honeywell Distributed ADA - Approach, Distributed ADA: Developments and Experiences, Proceedings of the Distributed Ada '89 Symposium, University of Southampton, Cambridge University Press.

13) Hutcheon A, Snowden D, Wellings A, Programming and Debugging Distributed Real-Time Applications in Ada, International Workshop on Real-Time Ada Issues, Moreton Hampstead, Devon, England, 13-15 May.

14) Arevalo S, Alvarez A, [1988], Fault Tolerant Distributed Ada, Ada Letters, Volume VIII, Number 7.

15) Bamberger J, Coddington T, Firth R, Stinchcomb D, Van Scoy R, "Kernal Architecture Manual", Technical Report CMU/SEI-89-TR-19, Distributed Ada Real Time Kernel, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, 1989.

16) Van Scoy R, Bamberger J, Firth R, "An Overview of Dark", Ada Letters, Volume IX, Number 7, pp. 91-101, November/December, 1989.

17) Bishop J.M., Hasling M.J., Distributed Ada - an Introduction, The Ada Companion Series, Cambridge University Press, 1989.

18) Allworth, S.T., Introduction to Real-Time Software Design, Springer-Verlag New York Inc., 1981.

19) Nielsen K., Shumate K., Designing Large Real-Time Systems with Ada, McGraw Hill, 1988.

20) Rogers P., Embedded/Realtime Systems in Ada Tutorial, Eighth Annual Washington Ada Symposium/Summer SigAda meeting, 17-21 June 1991.

21) Stallings W, "Local Networks", Computing Surveys, Vol 16, No.1, pp.4-38, March 1984.

22) Vector Editorial Staff, Moving towards MAP, Vector, pp.18-20, June 1989.

23) Chou Wushow, Computer Communications, Prentice-Hall, 1983.

24) Buhr R.J.A, System Design with Ada, Prentice-Hall, 1984.

25) Meridian Software Systems, The Meridian Ada 4.1 Compiler User's Guide PC Dos, 1991.

26) Novell Netware Application Programmers Interface, Novell Incorporated, 1988.

27) Novell Netware Getting Started: Supervisors Guide, Advanced Netware Version 2.15, Novell Incorporated, 1988.

28) Lee R.E., Advanced Netware Theory of Operations version 2.1, Novell Corporate Communications, 1987.

29) IBM Technical Reference PC Network, IBM Corporation, 1984.

30) Schwaderer W.D., C Programmers Guide to Netbios, Howard W. Sams & Company, 1988.

31) Schumate K., "Embedded Programming in Ada", course notes, 1989.

32) Barnes J., "An Introduction to Real-Time Programming", unpublished notes.

33) Ramseier G., Final Year Dissertation, Electronic Engineering, University of Natal, 1991.

34) Moodley K.A., Final Year Dissertation, Electronic Engineering, University of Natal, 1991.

35) Ada Board's Recommended Ada9X Strategy. Office of the Under Secretary of Defense for Acquisition, Washington D.C. 20301, 1988.

36) Ada9X Requirements, December 1990, Office of the Under Secretary of Defense for Acquisition, Washington D.C. 20301, 1990.