National Library of Canada | Bibliothèque nationale du Canada

Canadian Theses Service | Services des thèses canadiennes

Ottawa, Canada
K1A 0N4

CANADIAN THESES | THÈSES CANADIENNES

NOTICE

The quality of this microfiche is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy

Previously copyrighted materials (journal articles, published tests, etc.) are not filmed.

Reproduction in full or in part of this film is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30.

AVIS

La qualité de cette microfiche dépend grandement de la qualité de la thèse soumise au microfilmage Nous avons tout fait pour assurer une qualité supérieure de reproduction

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure

Les documents qui font déjà l'objet d'un droit d'auteur (articles de revue, examens publiés, etc) ne sont pas microfilmés

La reproduction, même partielle, de ce microfilm est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c C-30

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED

LA THÈSE A ÉTÉ MICROFILMÉE TELLE QUE NOUS L'AVONS REÇUE

NL-339 (r.86/06)

Canada

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-35507-7

Programming Language Support for Distributed Applications

Alan Jeffrey Madras

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of Master of Computer Science at Concordia University
Montréal, Québec, Canada

December 1986

© Alan Jeffrey Madras, 1986

ABSTRACT

Programming Language Support for
Distributed Applications

Alan Jeffrey Madras

The language d-Pascal was defined to provide high-level support for distributed applications programming, performing the functions of the Applications Layer of the ISO-OSI Reference Model [TANE 81]. Interprocess communication in d-Pascal is accomplished by the asynchronous exchange of messages. d-Pascal is described as a set of source language extensions to sequential Pascal [WIRT 71], and several annotated examples of d-Pascal programs are given. An implementation of d-Pascal is also described. A preprocessor that translates d-Pascal programs into sequential Pascal has been built, and the source transformations are described in detail. Operating system support required by the preprocessor is also discussed.

This thesis also includes a survey of other concurrent programming languages, categorizing them according to the model of interprocess communication used.

ACKNOWLEDGEMENTS

I wish to thank Dr. T. Radhakrishnan for his supervision of this thesis. His professional knowledge, his interest and patience, and his friendship during the course of my work were invaluable.

Thanks to Michel Devine, Clifford Grossner and John Opala for their proof-reading of this thesis. As well, discussions with Michel relating to languages and concurrency contributed greatly to my knowledge and appreciation of the subject.

The excellent research network upon which much of my work is based was artfully designed and constructed by Clifford Grossner.

I would especially like to thank my family for providing me with the support and the inspiration required to undertake this project.

I am indebted to Concordia University and the National Science and Engineering Research Council for their generous financial support.

TABLE OF CONTENTS

1.0 Introduction 1

2.0 An Overview of Concurrent Programming Languages 4
2.1 Models of Communication 5
2.2 The Procedure Call Model 8
2.2.1 Asynchronous Procedure Call Languages 10
2.2.2 Synchronous Procedure Call Languages 15
2.3 The Message Transfer Model 19
2.3.1 Synchronous Message Transfer Languages 21
2.3.2 Asynchronous Message Transfer Languages 23
2.4 The Process Call Model 27

3.0 Description of d-Pascal 31
3.1 LPS Declarations 33
3.2 System Variables 34
3.3 Message Handling Constructs 35
3.3.1 MESSAGE and WAIT 35
3.3.2 SEND and RECEIVE 38

4.0 An Implementation of d-Pascal 41
4.1 Working Environment 41
4.2 Implementation of d-Pascal Features 42
4.2.1 Constant and Type Information 43

4.2.2 LPS Declarations 44
4.2.3 System Variables 49
4.2.4 Message Handling Constructs 50
4.3 Programming In d-Pascal 53

5.0 Programming Examples 55
5.1 A Process for Merging 55
5.2 A Text Formatting Example 57
5.3 Data Abstraction 60
5.4 Quicksort 64

6.0 Summary 70

References 73

Appendix A. The Cuenet Facility for Distributed Computing 76
A.1 Distributed Computing Systems 76
A.2 Cuenet 79
A.2.1 The Components of Cuenet 79
A.2.2 Programmer-specified Configurability 83
A.2.3 How Messages are Transmitted 84
A.2.4 The Structure of a Cuenet Message 87

Appendix B. Syntax of d-Pascal Extensions 89

Appendix C. Sample Programs 90
C.1 The Merging Program 90

C.2 The Text Formatter Programs 91

C.3 The Server Program 95

C.4 The Quicksort Programs 97

LIST OF ILLUSTRATIONS

Figure 1. BNF definition of LPS declarations 33
Figure 2. BNF definition of MESSAGE and WAIT 36
Figure 3. BNF definition of SEND and RECEIVE 38
Figure 4. The translation of LPS declarations 46
Figure 5. The run time stack during a SEND 47
Figure 6. The run time stack during a RECEIVE 48
Figure 7. Translations of MESSAGE and WAIT 51
Figure 8. The translation of SEND and RECEIVE statements 52
Figure 9. Pseudocode for a Sequential Text Formatter 57
Figure 10. Pseudocode for the Concurrent Text Formatter 59
Figure 11. Pseudocode for the Server Process 61
Figure 12. Pseudocode for the main Quicksort process 65
Figure 13. Multicomputer Topologies 78
Figure 14. The Configuration of Cuenet 80
Figure 15. Message Communication Data Structures 85
Figure 16. Structure of a Cuenet Message 86

Chapter 1

INTRODUCTION

It is commonly said that "two heads are better than one." It is generally implied by this expression that two people may be able to solve a difficult problem in less time by cooperating than either person could working individually. Of course, if the problem is trivial, participation of the second "head" may slow down the solution process, as it may take longer for the first person to explain the problem to the second person than to solve it outright.

The foregoing analysis may also be applied to problem solving by computer. Several computers working together on a problem may be able to find a solution in less time than could any one of the machines by working alone, assuming that the intercomputer communication is not so great as to supersede the time spent working on the problem itself. Of course, the more compute-bound the algorithm relative to the amount of communication required, the greater the potential advantage of concurrent processing.

Programming concurrent systems is very difficult. In addition to all of the skills required for sequential programming, the concurrent programmer must also deal with nondeterminacy and, naturally enough, concurrency: many activities (processes) are executing concurrently, and each process may have to deal with external events, such as interaction with other processes, which occur at unpredictable times. Many concurrent programming languages have been developed to enable the programming of such systems. These languages must acknowledge the con-

cepts of processes, synchronization of processes, and communication between processes. That many concurrent languages were invented with operating systems applications in mind should not come as any surprise, as the twin problems of nondeterminacy and concurrency are the hallmarks of operating systems on both uniprocessors and multiprocessors. Chapter Two of this thesis presents a survey of some of the more representative concurrent programming languages. The survey attempts to introduce the major software concepts in concurrent programming, and show how they are used in a variety of languages. The different models of interprocess communication, message passing versus procedure call, are contrasted, as are synchronous versus asynchronous communication.

This thesis introduces d-Pascal, a language derived from sequential Pascal [WIRT 71], which offers the programmer high-level support necessary to write concurrent programs for a network of computers. d-Pascal supports processes which may communicate and synchronize with each other by exchanging messages across the network. It is intended to be used in research into distributed algorithms and program decomposition, areas of difficulty in the field of concurrent applications programming. The overriding goal in the design of d-Pascal has been to create an applications vehicle which was not so complex as to impede the programmer's ability to program applications, yet was sufficiently powerful to permit a range of applications wide enough to provide a viable environment for distributed algorithm research.

Chapters Three through Five describe d-Pascal. Chapter Three defines and discusses the concurrent programming features of d-Pascal; Chapter Four describes an implementation of d-Pascal, and discusses the

support required of the programming environment; Chapter Five contains some programming examples that demonstrate how the concurrent language features of d-Pascal are used.

d-Pascal has been implemented for this thesis on Cuenet, a loosely coupled network of microcomputers. As it is intended for algorithm research applications, d-Pascal falls within layer seven of the ISO Reference Model of Open Systems Interconnection (OSI) [TANE 81], the Applications layer. Session layer (layer five) and Presentation layer (layer six) issues are beyond the scope of d-Pascal, although the relationship between d-Pascal and the layer five and layer six requirements of the operating system is discussed in Chapter Four.

Appendix A describes Cuenet, the network upon which d-Pascal has been implemented. Appendix B gives a summary of the syntax of the d-Pascal concurrency features, and Appendix C contains the source code of the d-Pascal programs discussed in Chapter Five.

Chapter 2

AN OVERVIEW OF CONCURRENT PROGRAMMING LANGUAGES

Many concurrent programming languages have been proposed by language designers. In a concurrent program, several activities can be specified which may be executed concurrently. These activities are called **processes**.

Concurrency may be either real or apparent. With real concurrency, two or more processes are executing at exactly the same time, as opposed to apparent concurrency, in which execution switches back and forth among started processes, but only one is actually executing at any given instant. Real concurrency requires a different processor for each concurrent process, while apparent concurrency can be implemented on a single processor. See, for example, [LORI 72] for further discussion.

In the wide spectrum of concurrent programming languages, no single definition of a process will suffice in all cases. Consequently, none will be formally given here. Loosely speaking, in the context of this survey, a process can be considered to be part of a program which can run concurrently with other parts of the same program. The program itself can be anything from a very specific application to a general purpose operating system.

When several processes in a concurrent program cooperate in the solution of a problem, some interaction will generally be required to coordinate the problem solving effort. Process interaction may take the

form of synchronization, data exchange, or the exclusion of other processes while a shared resource is accessed.

In general, processes are not necessarily dependent on other processes for their existence, except if some master process is required to initiate all the other processes, such as in the language Modula [WIRT 77]. Processes live "a long time" as compared with procedures in block-structured languages. In fact, sometimes they live "forever" (for the lifetime of the job). Except when explicitly communicating, processes are generally not aware of the existence or activities of other processes.

Different models of interprocess communication will be presented in the next section, followed by a survey of some existing concurrent programming languages in the light of their concurrent facilities and constructs.

2.1 MODELS OF COMMUNICATION

Two themes for interprocess communication have come to dominate the discipline of concurrent programming language design. They are the Procedure Call Model and the Message Transfer Model [STAU 82]. Many variations on both themes exist, but most proposals can be classified according to one of the two models.

In a comparison of the models, J. Staunstrup [STAU 82] notes an orthogonal dimension: synchronous versus asynchronous communication.

When two processes communicate synchronously, the first of the processes to request the communication is suspended until the other makes the corresponding request. When communicating asynchronously, the processes interact with a medium (such as a monitor or message buffer), not with each other. Consequently, the processes need not necessarily wait for each other.

There exists a tradeoff between synchronous and asynchronous communication. Asynchronous communication offers the possibility of maximum parallelism, but it is less secure than synchronous communication because error conditions such as buffer overflow may occur in the medium between the communicating processes. When communicating synchronously, each process which participates in the communication can be certain that the other process participated as well, but the synchronization does inhibit concurrency to a certain degree.

Note that there are other definitions of synchronization which do not apply here. For instance, in [LINT 81], processes are considered to be synchronous if they run simultaneously (on different processors) and are controlled by the same processor clock, whereas asynchronous processes run with their own clocks, synchronizing only to communicate. The latter scheme is similar to the synchronous scheme of this survey.

Generally, a process can contain any number of procedures. In languages employing the procedure call model, procedures in a process can call procedures outside of the process. These external procedures are defined either in monitors [HOAR 74] or in other processes. The calling process is suspended until the called process completes execution of the called procedure. Languages employing monitors or monitor-like concepts

include Concurrent Pascal [BRIN 75], Edison [BRIN 81a], Concurrent Euclid [CORD 81] and Modula [WIRT 77]. These languages communicate asynchronously. Synchronous procedure call languages include Ada [ADA 81] and Distributed Processes [BRIN 78], where procedures in one process can directly call procedures in other processes.

A recent area of research in synchronous procedure call languages, intended mainly for processes executing on different computers, is the Remote Procedure Call (RPC) [BIRR 84]. RPCs are designed to look like regular (sequential) procedure calls to the programmer, but, because the called procedure is remote, the underlying implementation is similar to that of a distributed message passing system.

In languages using the message transfer model, a process communicates by sending data messages to other processes. The messages are structured in some format known to the communicating processes.

When communicating asynchronously, a process can send a message and continue executing, the receiving process accepting the message any time after it was sent. This implies some type of interprocess buffering system to hold messages until they are received. This method of communication is used in Platon [STAU 76], PLITS [FELD 79], "Communication Schemes" [JOSE 81] and d-Pascal, the latter being developed in this thesis.

In the synchronous version of the model, a sending process is suspended until the receiver is ready to receive the message. No buffering is provided, as messages are sent directly from one process to another.

Synchronous message passing is used in Communicating Sequential Processes (CSP) [HOAR 78] and CCS [MILN 80].

As with most seemingly reliable classification schemes, objects which cannot be classified are found. I will suggest a third model of communication which will I refer to as the Process Call Model, found in Pascal-C [LAM 82] and a proposal by Andre and Decitre [ANDR 78].

In the Process Call Model, the runtime process structure mirrors the procedure structure. Each concurrent process is a procedure designated to run in parallel with its calling procedure. This allows a process to initiate several slave processes to run concurrently. Slave processes can be nested. The lifetime of a process is the same as the lifetime of the procedure which makes up the process.

2.2 THE PROCEDURE CALL MODEL

The procedure call model is appropriate for applications in which the pattern of process interaction is similar to that of procedure calls. For example, if some process provides access to a resource, another process requiring the resource will request it, likely with arguments, and wait until the request is serviced before resuming execution. It is very convenient for the user to be able to code this type of interaction in the familiar form of the procedure call.

The kind of process generally used to field requests in the above example is the monitor [HOAR 74]. A monitor is a passive process containing private data, externally callable entries, a process queuing mechanism and an initialization section. A monitor is passive in the

sense that, after performing initialization operations on its private data, it only acts in response to external calls to one of its procedures. When no call is being serviced and no calls are pending, the monitor is quiescent. Note, however, that unlike blocks in a block-structured language, the monitor, including its data and procedures, continues to exist even when none of its procedures are executing.

It is essential to the integrity of a monitor's private data that only one external process be allowed to complete a call to a monitor procedure at any given time, for if more than one procedure body were executing simultaneously within a monitor, the effect on the local variables of the monitor would be unpredictable. Therefore, all requests must be suspended as long as one request is currently being serviced.

To this end, Hoare introduces a new type of variable known as a condition, and requires of the writer of the monitor to declare a condition variable for each reason why a process might have to wait.

A condition variable is not a true variable, in that it has no accessible value. It can be considered to be a queue of processes waiting on the condition. The queue, initially empty, is invisible to a process executing in the monitor.

Processes are delayed by a wait operation which causes the calling program to be delayed on a specified condition. A signal operation will cause exactly one program delayed on a specified queue to be resumed immediately, without the possibility of an intervening entry call from a third program. A signal on an empty condition has no effect.

Any appropriate queuing discipline can be used to schedule the waiting calls in such a way that all will be served eventually. Activating the longest waiting process is one such discipline.

Monitors are used in the asynchronous procedure call languages, as any communication between (active) processes takes place through the shared resource of a monitor. Calls to monitor entries are also asynchronous in the sense that the monitor procedures do not explicitly wait for a particular call: they passively wait for any anonymous call to occur.

In synchronous procedure call languages, processes call procedures in other (active) processes directly. A called process generally states that it is ready to accept an interprocess call to certain of its procedures in order for the call to complete.

The synchronous mechanisms in Distributed Processes and Ada will be discussed in "Synchronous Procedure Call Languages" on page 15.

2.2.1 Asynchronous Procedure Call Languages

The Hoare monitor concept is the dominant common feature of asynchronous procedure call languages.

The language Concurrent Pascal [BRIN 75] is derived from Pascal [WIRT 71], differing mainly in the addition of concurrent programming constructs called **processes**, **monitors**, **classes** and **queues**.

Concurrent Pascal was developed by Per Brinch Hansen of the California Institute of Technology during the years 1972 to 1975, and was intended for the structured programming of operating systems.

A Concurrent Pascal **process** consists of parameters, private data, and a sequential program to operate on them. One process cannot access the private data of another process. However, data may be shared if they are defined in a **monitor**, which is based on the Hoare monitor described above. To access a given monitor, a process requires **access rights** for that monitor. Access rights are given to a process through its parameter list when the process is initialized.

Concurrent Pascal monitors contain **queue** variables, which correspond to Hoare's condition variables. The operations **delay** and **continue** can be performed on queue variables, corresponding to Hoare's wait and signal, respectively.

A **class** is a system component which acts like a monitor in that it describes a data structure and the possible operations on it, but any given instance of a class can only be called by the process which declared it. Several processes can declare (local) instances of the same class. Classes serve mainly as a program structuring aid.

Modula [WIRT 77] is another concurrent language intended for operating system design. Like Concurrent Pascal, Modula is derived from (sequential) Pascal. Added are features for modular design, multiprogramming and device operation. But unlike Concurrent Pascal, Modula is intended primarily for programming dedicated computer systems. The designer of Modula, Niklaus Wirth, hoped to reduce the amount of assembly

language programming required to implement applications such as process control systems and input/output device drivers:

> "Assembly code is still used virtually exclusively in those applications whose predominant purpose is not to design a new system based on abstract specifications, but to operate an existing machine with all its particular devices.... A major aim of the research in Modula is to conquer that stronghold of assembly coding, or at least to attack it vigorously." [WIRT 77]

Modula attacks the problem by providing facilities for multiprogramming as well as facilities for operation of a computer's peripheral devices. To effect the latter, Modula allows the programmer access to the computer's device registers and interrupt facility, but encapsulates such machine-dependent items in constructs called **modules**, which are similar to Concurrent Pascal classes in that they allow only very restricted operations on their private data and are only callable from their defining process. (A type of module for concurrent programming will be discussed later in this section.) Modules are actually general purpose constructs, by no means restricted to localizing device-dependent operations. They are of central importance to Modula, and have lent the language their name (MODUlar programming LAnguage).

A Modula process looks like a procedure, except that it is executed concurrently with the program which initiated it. Processes can only be created by the main program. When control reaches the end of a process, the process ceases to exist.

Synchronization is achieved by the use of **signals**, which correspond to Hoare's conditions and Brinch Hansen's queues. Signals can be sent, and processes can wait for signals.

Processes cooperate via common variables. In Modula, the critical sections of a program are contained in **interface modules**, corresponding to Hoare's monitors.

Other similar languages have been proposed as well.

Concurrent Euclid [CORD 81, HOLT 83], designed by James Cordy and Richard Holt, is a language suited for implementing operating systems, compilers and specialized microprocessor applications. In [CORD 81], the authors state that Concurrent Euclid can serve as the basis for implementing software which is formally verifiable. This is because it is based on the programming language Euclid [LAMP 77], which was designed in 1976 as a language for developing verifiable software. It is based on Pascal, but several alterations were made to make programs more easily verifiable. For example, Euclid functions are prevented (by the compiler) from having side effects. As well, Euclid allows the programmer to insert assertions to aid in verification.

A subset of Euclid was extended with concurrent features based on monitors to give Concurrent Euclid.

The Edison programming language [BRIN 81a], designed by Per Brinch Hansen in 1980, is yet another language intended to perform the same sorts of tasks as the other languages discussed in this section, but the design philosophy is different [BRIN 81b]. Edison is a small language, derived from Pascal and Concurrent Pascal, which contains very few constructs.

In Edison, several concurrent processes can appear and disappear dynamically, as long as they appear and disappear at the same time. This

is performed using a cobegin statement (similar to Dijkstra's parbegin [DIJK 68]) which, along with modules (similar to Modula modules and a shared version of concurrent Pascal classes [BRIN 73]) and a when statement (a simple form of Hoare's conditional critical region [HOAR 72]), make up Edison's set of concurrent constructs.

Edison does not contain explicit monitors, but one can be coded as a module in which the procedure bodies contain single when statements. Monitors in Edison are the product of programmers style and discipline, not of language semantics enforced by the compiler. In [BRIN 81b], Brinch Hansen discusses his design decisions:

> "The result is a more flexible language based on fewer concepts in which one can achieve the same security as in Concurrent Pascal by adopting a programming style that corresponds to the processes and monitors of Concurrent Pascal. Or one can use the language to express entirely different concepts. On the other hand, it is also possible to break the structuring rules and write meaningless programs with a very erratic behaviour. I have adopted this more general and less secure approach to programming to learn from it. It is still too early to make firm conclusions about the consequences of such a compromise."

The principles of these languages are similar, as they are intended for roughly the same type of programming. Operating systems, whether general purpose or for process control, are very complex, and special languages and constructs which mirror the structure of operating systems greatly simplify their design and implementation. Operating system processes can be mapped onto process constructs in the language. Shared resources such as abstractions of peripheral drivers or file systems can be coded in monitors or interface modules. Concrete machine dependent representations of these abstractions can be coded into classes or device modules, where all implementation details not required by the pro-

cesses are hidden and protected, and only the names of the operations appear.

2.2.2 Synchronous Procedure Call Languages

In the synchronous version of the procedure call model, processes communicate directly with one another. They do not require the interposition of a passive "middleman process" of the monitor variety.

When an interprocess procedure call is made, the calling process is suspended until the called process is ready to accept the call. Of course, if the called process was already waiting for a call, the call will "connect" without delay. The calling process is then further delayed until the call completes.

The mechanisms for accepting procedure calls by a process vary among languages, but all must deal with the unpredictability of the order of external events. The ramifications in active processes are different than in passive processes. When a quiescent monitor process is ready to receive a call (that is, none of its procedures are active), it can accept a call to any of its procedures. Some service will be performed for the calling process, and the monitor will return to its quiescent state. Selection among waiting processes is purely a scheduling decision, the queuing discipline being determined at the time of implementation of the monitor. It is entirely beyond the control of the programmer using the monitor.

The situation in active processes, in which a program is generally running, is different. Some control must be exercised by the programmer

over which procedures can be called at different stages of the algorithm.

Most synchronous procedure call languages use some variation of Dijkstra's guarded commands [DIJK 75] as the basis for controlling nondeterminacy in a process' environment. A sequence of guarded commands is called a guarded command set. A guarded command set can appear in either an alternative construct or a repetitive construct.

A guarded command is composed of a guard and a guarded list. A guard is Boolean expression and a guarded list is a sequence of statements associated with a guard. A guarded list is eligible for execution whenever its guard is true.

When a guarded command set appears in an alternative construct, one of the guarded lists with a true guard will be selected for execution in a nondeterministic way. If none of the guards are true, the program aborts.

When a guarded command set appears in a repetitive construct, guarded lists with true guards will be selected one at a time in an unpredictable order as long as at least one of the guards is true. Guards are reevaluated between each execution of a guarded list. When a repetitive construct terminates, it is known that all of the guards are false.

Guarded commands themselves are not concurrent constructs, but they are well-suited for expressing and, to some extent, controlling the nondeterministic nature of parallel processing. Each guarded list can be coded to correspond to a procedure which may be called from another

process. The guards ensure that no procedure can be called unless the process is in an internal state suitable for the procedure's execution.

The potential for guarded commands to lead to formal derivation of programs through the development of a mathematically rigorous calculus as introduced in [DIJK 75] is not exploited in synchronous procedure call languages. Synchronous message passing languages rely much more heavily on guarded commands and therefore can be subject to rigorous mathematical examination. This will be discussed in the next section.

The programming language Ada [ADA 81] was designed by a team led by Jean Ichbiah in response to an initiative by the U.S. Department of Defense. Ada is a block-structured language intended for large real-time applications, with facilities for modular and concurrent programming.

Ada subroutines are modules in which algorithms are defined. They can be separately compiled. Types in Ada impose modularity on data structures. Packages are modules which specify collections of program and data resources. A task is a program unit very similar to a package, except that a task can run concurrently with other tasks. Tasks are the concurrent processes of Ada.

A task can contain entries. Externally, entries look like procedures, possibly with parameters. Other tasks can call entries by means of entry call statements. Within a task, the actions to be performed by one of its entries are specified in an accept statement. The actions are only executed when the called task is prepared to accept the entry call, i.e., reaches an accept statement. The acceptance of an entry

call is called a rendezvous. The calling task is suspended until the entry call is completed.

As well as providing a mechanism for synchronization and exchange of data through entry parameters, the rendezvous provides mutual exclusion, since, if two or more tasks call the same entry, only one call can be accepted at a time.

The **select** statement is a construct containing several accept statements. It allows a task to accept a call to any one of several entries. The accepts may each be associated with **guards**. A guard is a Boolean expression which must be true for its accept to be selected. The select statement with guards corresponds to Dijkstra's alternative construct.

A task will be suspended if it encounters an accept with an entry which has not been called, or a select with no entries (having true guards, if any) which have been called. A select with one of its entries called (or one entry with a true guard called) will not be suspended, the corresponding accept being immediately executed. If more than one entry (with true guards) in a select has been called, only one of the entries will be accepted, the choice being determined arbitrarily.

The concept of Distributed Processes [BRIN 78] was presented by Brinch Hansen as a basis for real time programming of distributed computing systems.

Distributed Processes are able to react to nondeterministic requests from an environment in which several things may be happening

simultaneously. Programs consist of a fixed number of concurrent processes which are started simultaneously and live for the duration of the application. Any process can only access its own data. Processes communicate solely by calling procedures in other processes. A process accepts a procedure call only when it is suspended while waiting for some condition to become true.

Nondeterminacy is controlled by two kinds of statements: the guarded command and the guarded region.

A guarded command in Distributed Processes has the same meaning as Dijkstra's guarded commands [DIJK 75]. There exists both the alternative form and the repetitive form.

Synchronization of processes is handled by means of nondeterministic guarded regions [BRIN 78a, HOAR 72] Guarded regions are similar to guarded statements, except that guarded regions will delay the process in which they occur as long as all its guards are false. The alternative guarded region will wait until one of the guards is true and execute the corresponding statements. The repetitive guarded region is an endless repetition of an alternative guarded region.

If several guards in a guarded command or region are true, one of them will be selected in an unpredictable manner.

2.3 THE MESSAGE TRANSFER MODEL

The procedure call model is appropriate for applications in which process interactions are bidirectional, where each communication re-

quires a response. In cases where communication tends to be unidirectional, such as in a producer-consumer framework in which explicit acknowledgements are not required, the message transfer model is called for.

In the message transfer model, processes communicate by sending and receiving messages. A message is simply information which is output by one process and input by one or more other processes. Messages travel over logical or abstract "links." The actual transferring of a message is generally not performed by the sending or receiving processes, but by some transparent serving process, such as message handling software in an operating system. To send or receive a message, a process simply places the message on or accepts the message from its representation of the interprocess link.

In the asynchronous version of the message transfer model, a process does not undergo any noticeable suspension in sending a message. (A process may be interrupted as the message handling software takes control of the message, but this transparent delay is not a language issue.) The message is sent, regardless of whether the destination process is ready to receive it. This implies that automatic buffering of messages must be provided by the runtime environment. The destination process can accept the message any time after it was sent.

The synchronous version of the model requires that the send operation and receive operation be completed simultaneously. The send operation will be suspended until a corresponding receive operation is initiated by the destination process.

In both the synchronous and asynchronous versions, a receive operation clearly must be suspended if there is no message ready to be received, but in neither version of the message transfer model is the sending process required to wait for the receiving process to issue a response after processing the message. The sending process is free to continue running as soon as the message is sent. This is the main semantic difference between the message transfer model and the procedure call model.

2.3.1 Synchronous Message Transfer Languages

In the article Communicating Sequential Processes (CSP) [HOAR 78], C. A. R. Hoare proposes a language for distributed computing. In CSP, input and output are treated as basic primitives of programming.

A CSP program consists of a fixed number of concurrently running processes with disjoint address spaces. This anticipates implementation on a moderately coupled network [FATH 83] of microcomputers. Architectural issues will be discussed in the next Chapter.

Communication and synchronization in CSP are accomplished through the input and output operations. For two processes to communicate, each must name the other, one in an input statement and one in an output statement. When this occurs, data is copied from the outputting (sending) process to the inputting (receiving) process. There is no buffering of messages. I/O to a physical device appears the same as interprocess communications, as devices are considered to be processes implemented in hardware rather than software. The value of the ex-

pression sent in an output operation must correspond in type to the variable in the matching input statement.

Sequential control and nondeterminacy in CSP are expressed entirely through Dijkstra's guarded commands, with minor changes in syntax. Input commands are permitted in guards. They are completed before the guarded list is executed. Extensions to CSP allowing output commands in guards have been proposed [HOAR 78, SILB 79].

Concurrent processes are created using a variation of Dijkstra's parbegin [DIJK 68]. All processes are created simultaneously, and the program generating the processes is suspended until all the processes finish. The concurrent processes cannot access variables belonging to any other process

As defined in [HOAR 78], CSP is not suitable for use as a programming language. The article presents only a partial solution to the communication and synchronization problems, concentrating on semantic issues and ignoring any implementation details, as well as keeping the notation extremely terse.

In fact, while the terseness of the language does not inhibit the expressive power of CSP, it anticipates rigorous mathematical analysis. Hoare states that the most serious problem ignored in the article is that of a proof method to assist in the development and verification of correct programs.

The proof issue is dealt with in "A Calculus of Communicating Systems" [MILN 80], in which Robin Milner presents a notation for describing communicating systems in a mathematically precise way. The calculus

is called CCS. CCS is not a programming language, but one can be derived from it. Milner states that CSP is similar to CCS, translations of algorithms from one to another being rather straightforward, and hopes that a semantics and proof theory for CSP can be developed from CCS.

Languages such as Euclid (and concurrent Euclid) that contain assertions supply redundant information to the compiler to allow partial verification of programs, but Euclid is much too large a language to permit complete mathematical treatment. CSP is small enough and close enough to a mathematical formulation to permit manual derivation of correct programs. Purely algorithmic derivation and verification of concurrent programs is still an open topic in research.

2.3.2 Asynchronous Message Transfer Languages

The distinguishing feature in asynchronous message transfer languages is the so-called "no-wait send". A process is free to send messages at any time, regardless of whether the receiver is ready to receive. After sending, the sender is free to continue executing.

Message buffering must be provided by message handling software external to the processes. The message handling software may or may not guarantee anything about the order in which messages are received relative to the order in which they were sent. If a language has such a requirement, the message handling software must fulfill it.

The versatility of asynchronous message communication is indicated by the diversity of concurrent languages which use it. Examples are the

PLITS system [FELD 79], which is intended for distributed applications with low density communication on a moderately coupled network, and Platon [STAU 76], a multiprogramming language for communications applications. The paper "Communication Schemes" [JOSE 81] by Mathai Joseph demonstrates how procedure call communication can be simulated with asynchronous message passing.

The PLITS system [FELD 79] was developed at the University of Rochester. PLITS is not a programming language, but a set of concepts which can be used to extend a sequential language. It combines modules, messages and assertions to provide a framework for concurrent programming.

In PLITS, processes are called **modules**. A module is a self-contained sequential program unit which runs in parallel with other modules. There is no common address space among modules.

Modules communicate with each other solely via **messages**. A message is a set of (name, value) pairs called **slots**. Slot names are declared in public declarations. To send a message, a module must name the appropriate public slot in the destination module. The value portion of a slot is an element of some primitive domain (such as integers) whose representation is generally understood. **Assertions** give additional information to the compiler to aid in program verification. They are beyond the scope of this thesis.

In PLITS, each module has its own buffer space, allocated and maintained by the operating system, to hold messages sent to, but not yet

requested by, the module. The programmer is not concerned with how or where a message is stored after it is sent.

The language Platon [STAU 76] is an asynchronous message passing language with quite a different philosophy from that of PLITS. Developed at the University of Aarhus, Denmark, by Staunstrup and Sorensen, Platon is derived from Pascal by introducing processes, shared variables and queue semaphores. Platon is primarily a multiprogramming language for the programming of devices such as terminal concentrators and controller simulators.

Platon processes can be declared and invoked in a nested fashion just like Pascal procedures. A given process can be invoked several times, each invocation running concurrently. A parent process can pass parameters to child processes.

In typical Platon applications, it is generally necessary for processes to exchange large amounts of data. Platon uses shared variables as the medium for interprocess communication. A shared variable can only be accessed by one process at a time. Access to shared variables is controlled by variables of types **reference** and **semaphore**.

The value of a variable of type reference is either a reference to a shared variable or nil. No shared variable can have more than one reference variable referring to it at any given time.

Processes exchange access to shared variables by means of variables of type **semaphore**, which are implemented as **queued semaphores** [LAUE 75], combining the synchronizing effect of Dijkstra's binary semaphores [DIJK 68] with the exchange of data. Queued semaphores can be con-

sidered to be mailboxes of unbounded capacity through which all interprocess communication must take place. Shared variables act as the envelopes in which messages are transmitted. Two standard operations, **wait** and **signal**, are used to exchange access to shared variables.

Another asynchronous message passing language is presented in "Communication Schemes" [JOSE 81] by Mathai Joseph. One of the main goals of the language is to support different schemes of communication as required by the application or the hardware without modification of the language.

The unit of concurrency is the module. Concurrently running modules can communicate with each other via ports. A message is sent through an oport and is received at the iports of one or more other modules. Each port declaration carries information on the data type of the message which can be sent through it, as well as a control rule for that port. When sending a message, the control rule for the oport determines whether the sender must wait for a response from the receiver or not. The control rule for an iport tells the receiver of a message whether or not a return message must be sent and whether the sender is suspended waiting for it.

A **connect** statement is provided to specify how oports of one module are to be connected with iports of other modules. Connected iports and oports must have compatible control rules. For example, an oport which suspends the sender until a responding message is received must be connected to an iport whose control rule compels the receiver to send a response.

The automatic response control rule is equivalent to a procedure call protocol. Thus this language can be used conveniently when either or both the message transfer or procedure call "communication schemes" are required by an application.

2.4 THE PROCESS CALL MODEL

Some languages, while supporting concurrency, do not support inter-process communication in the manner of either procedure call languages or message transfer languages. A new model, the Process Call Model, is proposed.

A striking feature of process call languages is their similarity to sequential block structured languages. Whereas many languages discussed in the previous sections were derived from Pascal, very few bore more than a passing resemblance to it, due to the addition of numerous major syntactic entities, such as modules, monitors, semaphores, etc. There are very few new syntactic constructs required to specify concurrency in process call languages. This is because of the unification of the concepts of process and procedure.

The language Pascal-C [LAM 82] was developed at Concordia University for solving combinatoric problems, where tree-structured searches requiring vast amounts of computation are common.

Pascal-C is derived from, and closely resembles, sequential Pascal. The unit of concurrency is the down procedure. It looks like a sequential procedure, except that its declaration is preceded by the word "down." (A down procedure is so named because, in typical Pascal-C

applications, calling a down procedure is essentially like descending one level in a tree-structured algorithm.) However, when called, a down procedure runs in parallel with its master (the process which called it), and is referred to as a slave process. A master can have any number of slaves, running the same or different down procedures. As well, any slave process can be a master to its own slaves. Slaves can receive call-by-value or call-by-reference parameters when invoked. Once running, a slave process cannot receive any more data from its master.

A set of slaves with a common master all running identical code is called a **process class**. A master can execute two types of statements on a process class. The **wait** statement suspends the master until all slave processes of a given process class are terminated. The **terminate** statement terminates all slave processes of the given class. This is the only way in which a master can affect a running slave in Pascal-C.

Return communication, from the slave to the master, can be accomplished in two ways:

1. while running, a slave can call a critical **procedure** in its master, or
2. upon reaching normal termination, the slave updates the actual parameters of its master with its call-by-reference parameters.

Updating can only occur when the master is executing a wait on the slave's process class.

A critical procedure can be called either by the process in which it is defined or by one of its slaves. Once invoked, it cannot be

interrupted. Critical procedures provide a way for slaves to invoke an action in their master.

Communication in Pascal-C has the form of procedure call communication, but functions like message transfer communication in that communications tend to be one way, not requiring an immediate response. The "no-wait call" of a down procedure gives a high potential for concurrency without introducing complicated new structures.

The simplicity of the "no-wait call" was also exploited by Andre and Decitre in their paper, "On Providing Distributed Application Programmers with Control over Synchronization" [ANDR 78]. Their application, accessing distributed data bases, is far removed from that of Pascal-C, but their aim is similar: to provide applications programmers with access to distributed systems without requiring them to learn unfamiliar high-level languages.

In their proposal, Andre and Decitre allow the programmers to invoke remote processes by naming them in a call to a special procedure INIT, whose parameters are:

- the location in the network of the process to be called;
- the name of the process to be called;
- a continuation; and
- any number of call-by-value parameters.

A continuation is the name of the procedure which should be called by the remote process to resume execution on the original calling site.

The remote process uses the INIT procedure to return its results by invoking the continuation procedure on the original site.

Process call languages are designed to run in distributed environments where the time for interprocess communication is of minor relative importance when compared with processing time.

This survey presented several concurrent languages, categorized by the model of interprocess communication used. d-Pascal, discussed in the following chapters, uses the asynchronous message passing paradigm.

Chapter 3

DESCRIPTION OF d-PASCAL

d-Pascal is a language for distributed applications programming. It is derived from sequential Pascal [WIRT 71], with additional constructs to implement communication and synchronization primitives. This Chapter describes the d-Pascal extensions only, and assumes the reader has knowledge of standard Pascal.

d-Pascal is an asynchronous message passing language. A d-Pascal application (called a distributed job, or d-job), consists of a set of program modules which are intended to execute simultaneously as concurrent processes. The processes are allocated statically and ~~start~~ executing at the outset of the d-job. Each process is assigned a **process ID** by the operating system. Processes may refer to each other by their process IDs.

d-Pascal assumes that the runtime environment will provide queues for incoming messages. The environment must queue a message before the d-Pascal process to which the message was sent can receive it. The runtime environment must keep a log of queued messages, which is accessible to d-Pascal primitives.

Messages in d-Pascal are sent and received through **logical ports** (l-ports). Each l-port has a type structure, and all messages are assumed to have the structure of the l-port through which they pass. The structure of an l-port is the concatenation of previously defined ab-

stract data types. Consequently, a message contains the concatenation of data values of appropriate types in the appropriate sequence.

It is possible for processes to exchange data with no processing by the runtime environment: the environment simply moves strings of bytes from origin to destination, without regard to the structure of the data. However, a sophisticated environment could perform two important functions with knowledge of a logical port's structure: type checking and data translation. Type checking is possible if the environment knows the structures of the origin and destination LPS's through which a message is to travel. If the LPS's are of different structure, the message transmission violates the type rules, and a runtime error occurs. Data translation would be an important function if d-Pascal modules within a d-job are intended to run on machines with different internal data representations. For example, if a message containing character data is to be sent from a computer with the ASCII character set to one with the EBCDIC character set, the runtime environment could perform the character translation. The sender would output characters in ASCII, and the receiver will receive them in EBCDIC. These functions are, however, beyond the scope of the application language.

To send a message (that is, data values), the application program must execute a SEND statement, specifying the destination of the message, the l-ports through which the message is to pass, and a sequence of expressions which evaluate to values of the types which make up the l-port (and the message). To receive a message, the program executes a RECEIVE statement, specifying the originating point of the desired message, the l-port through which it is to be received, and a list of vari-

ables, the types of which correspond to the structure of the l-port (and the message). The full description of d-Pascal communication and synchronization features follows.

3.1 LPS DECLARATIONS

```
<LPSDECL>     ::= LPS <LPSDEF> { ; <LPSDEF> } ; | empty
<LPSDEF>      ::= <LPS CONST> : <LPSMODE> : <LPSLIST>
<LPSMODE>     ::= IN | OUT | IN,OUT | OUT,IN
<LPSLIST>     ::= ( <ITEMS> )
<ITEMS>       ::= empty | <type ID> { , <type ID> }
<LPS CONST>   ::= <numeric one byte global CONST id or
                                        literal constant>
```

Figure 1. BNF definition of LPS declarations

All messages must be sent or received through **logical ports** (l-ports). The **mode** and **structure** of each logical port must be specified in the LPS declaration section, which begins with the key word **LPS** and follows the TYPE declaration section in the main block. "LPS" is an acronym for Logical Port Structure.

Each LPS declaration begins with an integral constant in the range 0..255. This constant, the **logical port number** (LPN), is the sole means of referencing an l-port.

The **mode** of a logical port determines whether the l-port is to be used to send or receive messages. The modes are named OUT and IN, respectively. The **structure** of a logical port is the concatenation of

any number of previously defined data types. The structure of an l-port is declared by simply listing the names of the required types in the desired sequence.

Two l-ports may have the same LPN only if they are of different modes. If two l-ports, one IN and one OUT, have the same structure, they can be declared simultaneously, with the mode specified as IN,OUT or OUT,IN. Any number of logical ports may have the same structure. The syntax of the LPS declaration is given in Figure 1 on page 33.

An l-port is said to be local to a module if it is declared within that module; otherwise, it is external. All local l-ports must be uniquely identifiable by their LPN and mode, but LPNs need not be unique between processes.

3.2 SYSTEM VARIABLES

A **system** variable is a variable which carries information about a message which has been queued by a process' operating environment, waiting to be received by the process. It is denoted by an ampersand (&) followed by a system variable name. d-Pascal defines two system variables, &SENDER and &LPORT. They are set when the MESSAGE function is called and returns the value TRUE, or when the WAIT statement terminates. They are of type 0..255. (See "MESSAGE and WAIT" on page 35 for the descriptions of MESSAGE and WAIT.)

A system variable can be used anywhere a user-defined variable of the same type is allowed. System variables are global within a module.

3.3 MESSAGE HANDLING CONSTRUCTS

Four extensions are available for handling messages. In various combinations these low-level constructs can be used to implement a wide range of high-level communication and synchronization schemes.

Recall that a message is said to be **queued** if it has been received by the runtime environment but not (yet) requested by the process to which it was sent. d-Pascal processes assume that the messages are queued in the order in which they were received by the environment, and, more significantly, that two messages sent from the same origin to the same destination are queued by the receiver's environment in the order in which they were sent, although there may, of course, be intervening messages from other sources. After a queued message has been requested by a d-Pascal RECEIVE statement (described in "SEND and RECEIVE" on page 37), it is deleted from the environment's queue.

A queued message can be identified by two characteristics: the process ID of its originator and the local l-port through which it was received. The application programmer makes use of these identifying characteristics when using the message handling constructs described in this section. If more than one message bearing identical characteristics is queued, the first one queued is the one selected by the environment.

3.3.1 MESSAGE and WAIT

```
<MESSAGE>      ::= MESSAGE ( <ARGLIST> )
<WAIT>         ::= WAIT ( <ARGLIST> )
<ARGLIST>      ::= empty | <LPORT ARG> | <LPORT ARG> , <SENDER ARG> |
                       <SENDER ARG> | <SENDER ARG> , <LPORT ARG>
<LPORT ARG>    ::= LPORT = <expression>
<SENDER ARG>   ::= SENDER = <expression>
```

Figure 2. BNF definition of MESSAGE and WAIT

The MESSAGE construct is a Boolean function which can be called with zero, one or two arguments, and which has side effects affecting the system variables. It is used to enquire of the slave OS about messages that are queued. Two Boolean conditions are associated with the MESSAGE construct. They are AND'ed together to give the result. Each condition corresponds to a possible argument to the MESSAGE function. None, one or both of the arguments may be present in the call, but both conditions are always evaluated.

The possible arguments of the call are the SENDER and LPORT arguments. If the SENDER argument is present, the corresponding condition can be TRUE only if a queued message has been sent by the process whose process ID is equal to the value of the SENDER argument. If the SENDER argument is absent, a queued message from any process will satisfy the sender condition. If the LPORT argument is present, the LPORT condition can only be satisfied if a queued message was sent to a local l-port whose LPN is equal to the value of the argument. If the LPORT argument is absent, a message received at any l-port will satisfy the condition. If both conditions are met, MESSAGE returns the value TRUE, and the system variables are set to values appropriate to a message which fulfilled

the MESSAGE conditions: &SENDER is set to the process ID of the sender of the message, and &LPORT is set to the LPN of the l-port through which the message was received. If either condition is not met, MESSAGE returns FALSE and the system variables remain unchanged.

For example, consider the Boolean-valued expression:

```
message(lport=REQUEST)
```

The LPORT argument is present, the SENDER argument is absent. If, at the time the above call was made, a message is queued which was received through the l-port whose LPN is equal to the value of REQUEST, irrespective of the source of the message (the SENDER argument is absent, any sender will satisfy the SENDER condition), the value TRUE is returned and the system variables &SENDER and &LPORT are set to the process ID of the originator of the message, and the LPN of the l-port to which the message was sent, respectively. If no message, received through l-port:REQUEST, is queued at the time of the call to MESSAGE, FALSE is returned and the system variables remain unchanged.

The WAIT construct is very similar to the MESSAGE construct, in that it has the same conditions and the same effects on the system variables as MESSAGE. However, instead of returning the value of the conditions, WAIT is a statement which suspends execution of the process until both conditions become TRUE.

MESSAGE and WAIT implicitly support nondeterminism, as they permit messages of unspecified types, or sent by unspecified senders, to be accessed.

3.3.2 SEND and RECEIVE

```
<SEND>       ::= SEND TO <expression> AT <expression> VIA
                       <expression> : ( <SENDLIST> )
<SENDLIST>   ::= empty | <expression> { <BAR> <expression> }
<RECEIVE>    ::= RECEIVE FROM <expression> VIA <expression> :
                       ( <RCVLIST> )
<RCVLIST>    ::= empty | <identifier> { <BAR> <identifier> }
<BAR>        ::= <the vertical bar character>
```

Figure 3. BNF definition of SEND and RECEIVE

The SEND statement is used to transfer data between processes. When sending data, the application must provide the number of the external l-port (belonging to the destination process) to which the data is to be sent (TO clause), the process ID of the destination process (AT clause) and the number of the local l-port through which the data is to be sent (VIA clause). All clauses must be present. The data to be sent are given as a list of expressions whose values are of the types declared for the LPS named in the VIA clause. The expressions are separated by the vertical bar (|) character.

The RECEIVE statement is used to acquire data sent by another process. To receive data, the application program must specify the process ID of the sender of the message (FROM clause) and the number of the local l-port to which the message was sent (VIA clause). The latter must be the number of an l-port declared to be of IN mode. A list of variable names is given. The variables will be assigned the values contained in the fields of the message. The variables must correspond in

number and in type to the list of types given in the LPS declaration of the l-port through which the data is to be received.

The SEND statement is executed without blocking the sending process. It is not necessary for the intended receiver to execute a RECEIVE statement for the message to be sent, as the receiver's slave OS will queue the message until it is requested. (Note that a totally non-blocking send may be impossible to implement, as the receiver's physical message buffer must be of a finite size: if the buffer is exceeded, either messages will be lost, or the run time environment will be forced to intervene and suspend the sending process until there is space in the receiver's buffer.) In contrast, the RECEIVE statement will execute without delay only if a message from the specified sender, sent to the specified l-port, is already queued, waiting to be requested. Otherwise, the RECEIVE statement will suspend the process executing it until an appropriate message is queued by the runtime environment.

One effect of the non-blocking SEND is that the sending process has no direct way of knowing when its message is actually received at its destination. The programmer may implement message acknowledgement by declaring an IN l-port specifically for this purpose, and including its LPN in the outgoing message which is to be acknowledged. When the destination process receives the message, it sends a message back to the designated l-port of the original sender. Note that this return message need not contain any data. The following sections of code, belonging to the sending process, illustrate this concept:

```
lps
   OUTGOING : out : ( ... , byte );   (* l-port for message
                                      to be acknowledged *)
   ACK      : in  : ();                   (* no fields *)
   .
   .
   .
   send to EXTLPN at OTHER via OUTGOING : ( ... | ACK );
   receive from OTHER via ACK : ();
   .
   .
   .
```

where:

EXTPLN	is the LPN of the external l-port to which the message is being sent;
OTHER	is the process ID of the receiving process;
OUTGOING	is the LPN of a local OUT l-port;
ACK	is the LPN of a local IN l-port; and
...	is the list of expressions containing the data in the message which is to be sent and acknowledged.

Note that it is possible for an l-port to not have any fields. The reception of a message through such an l-port is significant even though the message does not contain any data.

Chapter 4

AN IMPLEMENTATION OF d-PASCAL

4.1 WORKING ENVIRONMENT

d-Pascal has been implemented on Cuenet [GROS 82, RADH 85], a loosely-coupled network of microcomputers. Cuenet is described in detail in "Appendix A. The Cuenet Facility for Distributed Computing" on page 76. This section introduces Cuenet as a site for the implementation of d-Pascal.

Cuenet is intended for research in general distributed programming applications. One of the computers on the network is specially designated to be the master processor; the remainder are slave processors.

The function of the master is to act as the interface between the user and Cuenet's computing resources. The master handles all user requests, such as the creation, editing, storing and cataloguing of files, compiling, loading, linking and running of programs, and the reporting of statistics on the user's jobs. The master may invoke slaves to carry out the actions. The master is also responsible for allocating slaves to the user for running distributed jobs (**d-jobs**).

A typical d-job consists of several modules. Each module is a complete, independently compiled (or assembled) executable program. The running of a d-job is the simultaneous running of each of its component modules. Concurrently running modules (**processes**) are able to communicate with each other by exchanging **messages**. A message on Cuenet is a

sequence of bytes. The first few bytes make up the header, and the remaining bytes comprise the text of the message. The header contains information such as the source and destination of the message, sequencing information, and possibly other details. The precise physical description of a Cuenet message is given in "Appendix A. The Cuenet Facility for Distributed Computing". It is not of great importance to the applications programmer, as d-Pascal contains high-level constructs which provide the runtime environment with the information required to construct and send, or receive and interpret, a Cuenet message.

The main function of a Cuenet slave computer is to be a processing site for a module. Each slave has an operating system called a **slave operating system** (SOS), which is generally a standard operating system extended for use with Cuenet. Each SOS has the capacity to send messages to other SOS's, to receive messages from other SOS's, and to queue messages until they are requested by an application process. All messages are generated by the d-job. SOS's do not generate messages, they simply effect their transmission and reception. In fact, the slave operating system on Cuenet performs most of the functions required of the runtime environment, which are functions of layers 5 and 6 of the ISO/OSI Reference Model.

4.2 IMPLEMENTATION OF D-PASCAL FEATURES

A preprocessor has been written which translates d-Pascal into P-6800 Pascal [GIBB 81], a dialect very similar to "standard" Pascal [JENS 74]. The preprocessor is written in P-6800 Pascal, and runs on

the Motorola 6809 microcomputer. Non-standard P-6800 features used by the preprocessor will be described in this Chapter.

The input to the d-Pascal preprocessor is a d-Pascal program; the output is P-6800 Pascal program. The former will be referred to as the preprocessor source, while the latter will be referred to as the compiler source. Any statement in the preprocessor source which does not contain any d-Pascal extensions will be output to the compiler source unchanged. d-Pascal extensions will be translated to P-6800 Pascal equivalents before being output to the compiler source. This Chapter describes these translations, and the way in which the resulting Pascal program interacts with its environment to implement the d-Pascal features.

4.2.1 Constant and Type Information

The preprocessor maintains a symbol table of all the identifiers declared in the CONST and TYPE declaration sections of the main program. For each CONST declaration, the value of the constant and the number of bytes required to store it are recorded. For each TYPE declaration, the number of bytes required to store a value of the declared type is determined and recorded. For example, given the declarations:

```
const
  A = 1.5;
type
  V = array[1..40] of byte;
  R = record
        F1,F2 : real;
        F3    : V
      end;
```

the symbol table for a P-6800 preprocessor would contain the following entries:

```
A . . . . . . . 5 . . constant
BYTE . . . . . 1 . . type
R . . . . . . . 50 . . type
REAL . . . . . 5 . . type
V . . . . . . . 40 . . type
```

(The type BYTE is predeclared in P-6800 Pascal as 0..255. A REAL value requires five bytes of storage.) Other predeclared type and constant identifiers (ALFA, CHAR, TRUE, etc.) will also appear in the table. Note that record field identifiers are not included, as they do not represent a data type.

The number of bytes of data represented by each type is important because the preprocessor must be able to determine the size of each l-port declared in the preprocessor source program.

4.2.2 LPS Declarations

An l-port may be considered to be a window through which messages of a particular structure pass in order to reach other modules. However, d-Pascal does not describe the nature of the intermodule medium. In the Cuenet environment, as the Slave Operating System effects the transfer of messages, there must be a mechanism to transfer the message data, as well as associated data such as addressing information, to the SOS.

The current implementation of d-Pascal requires two pieces of object code which must be part of the SOS's message handling software: one

for executing SEND's, and one for executing RECEIVE's. This Chapter refers to them as SOS SEND and SOS RECEIVE, respectively. The d-Pascal preprocessor must know the addresses at which these programs will be loaded when the compiled modules will eventually be run.

The preprocessor translates each LPS declaration into an EXTERNAL procedure declaration in the compiler source file,[1] but the external addresses for all procedures corresponding to IN l-ports are the same (the address of SOS_RECEIVE), and the addresses for procedures corresponding to OUT l-ports are all the address of SOS_SEND. IN,OUT (and OUT,IN) LPS declarations generate two procedure declarations, one for the IN l-port, and one for the OUT l-port. As will be described in "SEND and RECEIVE" on page 51, the d-Pascal SEND and RECEIVE statements are translated into calls to the appropriate procedure heading. Thus, the translation of, say, a SEND statement, which is a procedure call, causes the arguments of the call to be placed on the Pascal run time stack using the familiar Pascal parameter passing methods, and control is then passed to SOS_SEND, which is able to read data from the same program stack, thus accessing all the data required to effect the SEND operation.

The procedure names used in the external declarations are constructed from the LPN and the mode of the l-port. Figure 4 on page 46 illustrates an example of LPS translation.

---

[1] P-6800 Pascal allows separate compilation of routines. External procedures must be declared EXTERNAL by the calling program, and their load address must appear in the declaration.

```
lps
    5 : out : (byte);
    6 : in  : (integer, real);
```

The above LPS declarations will be translated to the following procedure declarations:

```
procedure ZZZZO5(ZZZRPI, ZZZRLP, ZZZNUM : byte;
             ZZZ1 : byte);
    external $XXXX;

procedure ZZZZI6(ZZZSPI, ZZZRLP, ZZZNUM : byte;
             var ZZZ1 : integer; var ZZZ2 : real;
             ZZZA, ZZZB : byte);
    external $YYYY;
```

Figure 4. The translation of LPS declarations: The strings $XXXX and $YYYY represent the addresses of SOS_SEND and SOS_RECEIVE, respectively.

4.2.2.1 Implementation of an OUT 1-port

The code addressed by a declaration such as that for ZZZZO5 in Figure 4 is the SOS assembly language program referred to as SOS_SEND. When invoked, SOS_SEND has access to the P-6800 run time stack, as P-6800 Pascal stores the stack pointer at a known address. The data on the stack will be as diagrammed in Figure 5 on page 47, showing the following parameters:

ZZZRPI the process ID of the destination process, used by the SOS to address the physical message.

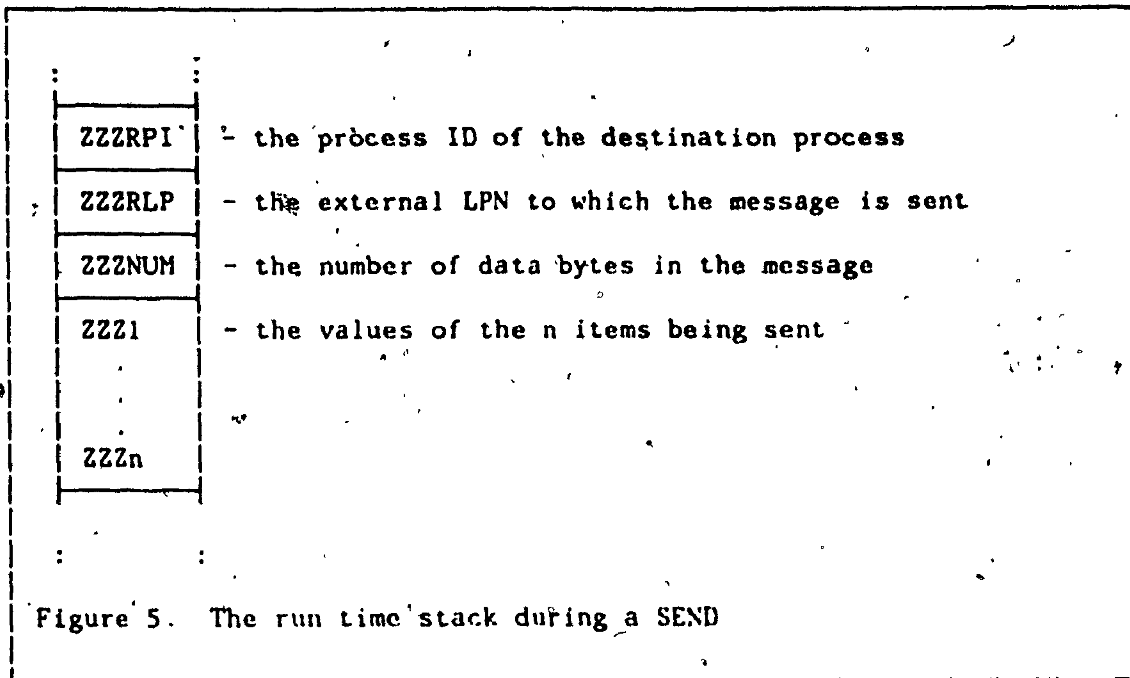

Figure 5. The run time stack during a SEND

ZZZRLP the number of the external logical port to which the message is being sent. The receiving SOS includes this information in its log of received messages.

ZZZNUM the number of data bytes in the application message. The preprocessor is able to calculate this number from the types named in the LPS declaration, and the Pascal compiler can ensure its correctness through strong type checking, as discussed in "Programming In d-Pascal" on page 53.

ZZZ1...ZZZn the values of the data items constituting the application message. The types of the ZZZi parameters in the procedure declaration are the same, in sequence and number, as the types listed in the LPS declaration.

When called, the program SOS_SEND creates the text of the message by simply copying ZZZNUM bytes from the program stack to the output message area.

4.2.2.2 Implementation of an IN l-port

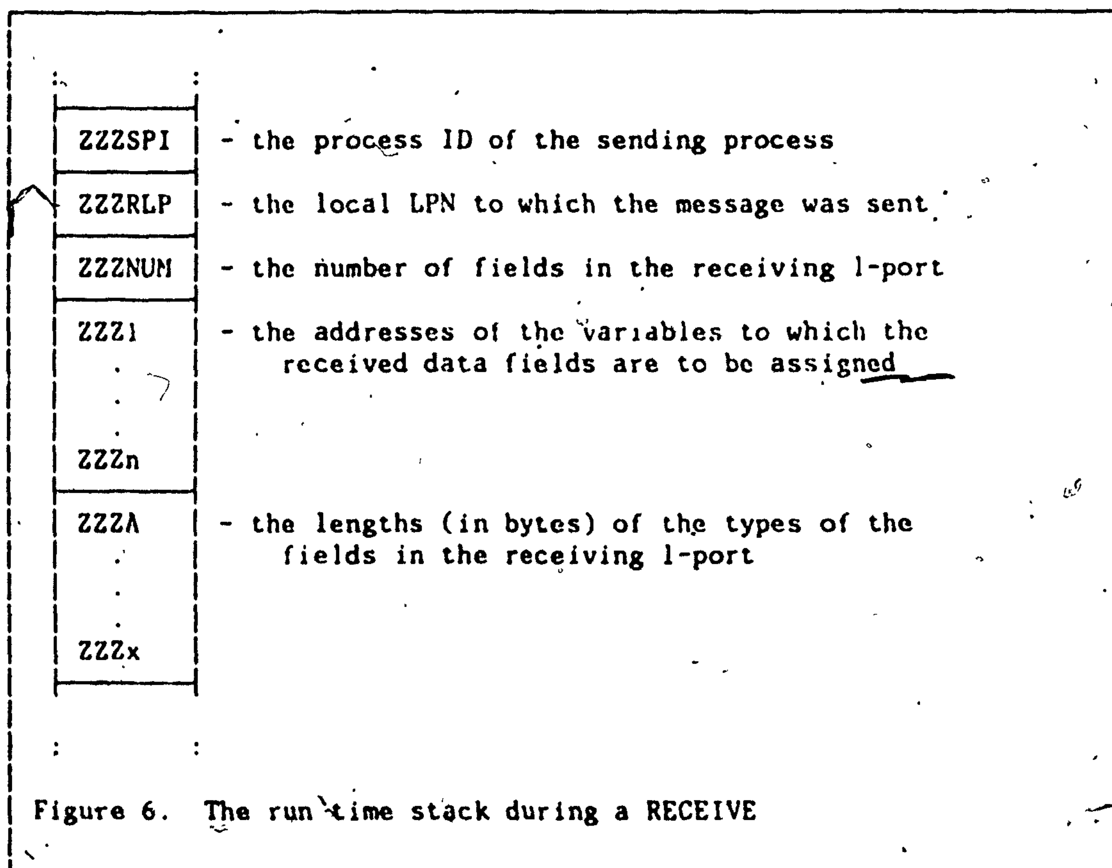

Figure 6. The run-time stack during a RECEIVE

When SOS_RECEIVE is called, the program stack should be configured as shown in Figure 6, containing the following values:

ZZZSPI the process ID of the process which originated the message.

ZZZRLP the LPN of the local IN 1-port to which the message was sent.

ZZZNUM the number of fields in the receiving 1-port (not the number of bytes, as in SOS_SEND).

ZZZ1...ZZZn the addresses of the variables to which data received through the fields of the 1-port are to be assigned. Note in Figure 4 that these variables are call-by-reference.

ZZZA...ZZZx the lengths of the values which are received through the 1-port, in bytes.

SOS_RECEIVE loops ZZZNUM times, copying ZZZA bytes from the data portion of the message to memory starting at address ZZZ1, ZZZB bytes to memory starting at address ZZZ2, etc.

4.2.3 System Variables

In the present system, system variables are implemented as variables declared in the main block of the compiler source with P-6800 Pascal's $A=... compiler option ([GIBB 81], p. 3-3).[2] When setting a system variable, the slave operating system need only set the appropriate memory location (when MESSAGE returns TRUE).

---

[2] The $A=... option forces variables to be stored at a fixed, static address specified in the source code.

The system variables &SENDER and &LPORT are named ZZZZS and ZZZZL, respectively, in the compiler input file.

4.2.4 Message Handling Constructs

The MESSAGE predicate is translated into a function call, the WAIT statement into a REPEAT loop, and SEND and RECEIVE are translated into procedure calls. The significance of the arguments are given in this section.

4.2.4.1 MESSAGE and WAIT

After the translations of the LPS declarations into procedure headings, the preprocessor generates a declaration for an external function named ZZZZM. ZZZZM accepts two BYTE arguments, and returns a Boolean result.

The translation of the MESSAGE construct is a call to the function ZZZZM. The first argument of the call is the string of tokens, copied character by character, making up the expression to be evaluated for the SENDER condition. If the condition is absent, the argument is the constant -1. The second argument of the call is the expression to be evaluated for the LPORT condition, or -1 if the condition is absent.

```
B1 := message;
B2 := message(lport=1);
wait(lport=&LPORT,sender=N+M);
```

The above d-Pascal statements are translated by the preprocessor to the following Pascal source:

```
B1 := ZZZZM(-1,-1);
B2 := ZZZZM(-1,1);
repeat until ZZZZM(N+M,ZZZZL);
```

Figure 7. Translations of MESSAGE and WAIT

WAIT is simply translated into "REPEAT UNTIL ZZZZM(...)," which can be thought of as "wait until MESSAGE returns TRUE."[3] See Figure 7 on page 51 for examples of translations of calls to MESSAGE and WAIT.

4.2.4.2 SEND and RECEIVE

The SEND and RECEIVE statements are translated into calls to the external procedure headings discussed in "LPS Declarations" on page 44. The name of the procedure called is made up of 'ZZZZ', an 'O' for SEND or an 'I' for RECEIVE, and the number of the local LPS specified in the VIA clause of the SEND or RECEIVE statement. Figure 8 on page 52 shows

---

[3] This primitive busy-wait implementation is suitable for the present Cuenet environment, as the SOS does not support multitasking.

the translations of SEND and RECEIVE statements which could send and receive messages through the l-ports shown in Figure 4 on page 46.

```
send to EXTLPN at PROCESS_ID via 5 : (N+127);
receive from PROCESS_ID via 6 : (INT_VAR | REAL_VAR);


The above d-Pascal statements will be translated
to the following procedure calls:


ZZZZ05(PROCESS_ID,EXTLPN,1,N+127);
ZZZZI6(PROCESS_ID,6,2,INT_VAR,REAL_VAR,2,5);
```

Figure 8. The translation of SEND and RECEIVE statements

In the translation of the SEND statement, the first argument is the logical module number of the intended receiver, inferred by the preprocessor from the AT clause. The second argument is the number of the external l-port (at the receiving site) to which the message is to be sent (TO clause). The third argument is the total number of bytes in the message, calculated by the preprocessor when parsing the corresponding LPS declaration. The next arguments are the expressions given in the SEND statement, which are listed between parentheses and separated by the vertical bar (|) in the application program. The preprocessor need not evaluate the expressions: it just copies them verbatim.

In the translation of the RECEIVE statement, the first argument is the logical module number of the sender of the message (FROM clause). The second argument is the number of the local LPS through which the message is to be received (VIA clause). The third argument is the num-

ber of fields in the message to be received (determined while parsing the LPS declarations), say, n. n arguments follow, the names of variables into which the data in fields of the message are to be stored, which are obtained from the list in the RECEIVE statement. n more arguments follow, the lengths in bytes of the data types of each of the fields in the l-port.

4.3 PROGRAMMING IN D-PASCAL

Coding in d-Pascal is similar to coding in sequential Pascal, as the languages are very similar. Pascal is a strongly typed language, and strong typing is evident in the d-Pascal extensions as well, with some exceptions. The use of Pascal procedure call facilities to transfer message data between the application program and the SOS during a SEND operation ensures type conformability between a message and its OUT l-port, but no such checking can be done when a message is received, as received bytes are transferred to the storage locations of the receiving application's variables outside of Pascal's jurisdiction, i.e. by SOS_RECEIVE. The current version of the slave operating systems on Cuenet provide no assistance to the programmer on this issue, but the d-Pascal preprocessor is capable of providing sufficient information to permit the enhancement of the operating system to check types of messages between modules.

The preprocessor produces a file containing information on the fields in the l-ports of a module as it translates the module to P-6800 Pascal. The SOS of a module which originates a message could send the information about the structure of the OUT (sending) l-port to the re-

ceiving SOS, which can compare it to the structure of the IN l-port to which the message was sent. If the l-ports are conformable, then the SEND statement and the RECEIVE statement in the communicating modules are also conformable.

The preprocessor scans the preprocessor source looking for "interesting" constructs, such as CONST and TYPE declarations and d-Pascal keywords. Standard Pascal statements are copied to the compiler source verbatim. The preprocessor source is parsed in recursive descent, and extensive syntactic diagnostics are issued for all extended statements, but, of course, the bulk of source code errors are detected by the Pascal compiler, and thus relate to line numbers in the compiler program listing. To facilitate the matching of a compiler source line with the preprocessor source file, the preprocessor writes a comment to each line of the compiler source file, indicating from which line in the preprocessor source file the compiler source line was generated. As an additional aid to debugging, the statement indentation in the preprocessor source is also maintained as much as possible in the compiler source file.

Chapter 5

PROGRAMMING EXAMPLES

Asynchronous message passing is an extremely versatile paradigm for distributed computing, in that programmers can simulate other communication schemes by following certain disciplines while programming an application. Most trivially, synchronous message passing may be simulated by always waiting for an acknowledgement just after sending a message, thus synchronizing the communicating processes by simulating blocking sends. This, of course, is not the true synchronous model, as message buffers are still required.

The programming examples in this Chapter have been chosen to highlight d-Pascal communication features and indicate some of the requirements for interactions between d-Pascal and the slave operating systems.

5.1 A PROCESS FOR MERGING

To illustrate the SEND and RECEIVE statements, a simple merge process is presented. The process inputs two streams of integers, both assumed to be strictly ascending, and outputs a merged list of these values. Any value that appears in both input streams is not duplicated in the output stream, and each input stream is terminated by the value MAXINT, which is also used to terminate the output stream.

The merge process will assume two producing processes P1 and P2, and a consuming process, C, with input l-port 1. Integers will be received from P1 and P2 through an IN l-port STREAM, and sent to C through

an OUT l-port of the same structure. The two l-ports are declared simultaneously:

```
lps
   STREAM : in,out : (integer);
```

(These two l-ports may be pictured as one bidirectional l-port, although, technically, this is not the case.)

The merge algorithm is coded in d-Pascal:

```
receive from P1 via STREAM : (V1);
receive from P2 via STREAM : (V2);

DONE := FALSE;
repeat
   if V1<V2 then begin
      send to 1 at C via STREAM : (V1);
      receive from P1 via STREAM : (V1);
   end
   else if V2<V1 then begin
      send to 1 at C via STREAM : (V2);
      receive from P2 via STREAM : (V2);
   end
   else if V1<>MAXINT then begin
      send to 1 at C via STREAM : (V1);
      receive from P1 via STREAM : (V1);
      receive from P2 via STREAM : (V2);
   end
   else   (* V1=V2=MAXINT *)
      DONE := TRUE;
until DONE;
```

Recall that the RECEIVE statement will block if no message from the specified process has been received through the specified logical port by the SOS.

The complete d-Pascal code for the MERGE process is given in "Appendix C. Sample Programs".

5.2 A TEXT FORMATTING EXAMPLE

Consider a text formatter which is capable of hyphenating words at the ends of output text lines as required. The sequential algorithm could be as shown in Figure 9.

```
program FORMATTER

   read WORD from INPUT_FILE
   initialize CURRENT_LINE = ''

   loop until end of INPUT_FILE
      if WORD fits on CURRENT_LINE then
         add WORD to end of CURRENT_LINE
      else
         try to hyphenate WORD into FIRST_PART and SECOND_PART
         add FIRST_PART to end of CURRENT_LINE
         format CURRENT_LINE
         write CURRENT_LINE to OUTPUT_FILE
         initialize CURRENT_LINE = SECOND_PART
      end if

      read WORD from INPUT_FILE

   end loop

   write CURRENT_LINE to OUTPUT_FILE

end
```

Figure 9. Pseudocode for a Sequential Text Formatter

Assume that the hyphenation routine requires two inputs, the word to be hyphenated and the maximum number of letters which may by put at the end

of the current line. The two outputs of the routine are the two parts of the split word. For simplicity, assume that the first part is returned with a hyphen appended to it. If the hyphenator is unable to split a word, it returns the null string as the first part, and the entire word is returned as the second part.

Two activities in the formatter's loop are candidates for removal to concurrent processes: word hyphenation, and line formatting and output. Say the text formatter determines that the next input word will not fit on the current line. It can send that word to the hyphenating process, and start reading words for the next line. Of course, while counting spaces on the next line, the formatter must allow for the possibility that the hyphenator will be unable to split its word, and that the whole word will have to be placed at the beginning of the next line. In any case, as soon as the hyphenator returns its result, the line just completed can be sent to the output process for final formatting (e.g. blank insertion for justification) and output.

The concurrent formatter algorithm is shown in Figure 10 on page 59. The control structure is somewhat more complex than in the sequential algorithm. Two line buffers are required, a flag is required to determine whether the previous line is waiting for the results of the hyphenation process, and the test "if WORD fits on CURRENT_LINE" must take into account the maximum number of characters which may be returned from the hyphenation process to SECOND_PART. (The control should be slightly more complex than shown, as the test for WORD fitting on CURRENT_LINE should be repeated after the results of the hyphenation process are obtained, but this detail has been omitted so as not to ob-

```
process CONCURRENT_FORMATTER

   read WORD from INPUT_FILE
   initialize CURRENT_LINE = ''

   loop until end of INPUT_FILE
      if WORD fits on CURRENT_LINE then
         add WORD to end of CURRENT_LINE
      else
         if a hyphenation attempt is pending
            RECEIVE result of pending hyphenation
            add FIRST_PART to end of PREVIOUS_LINE
            SEND PREVIOUS_LINE to format/output process
            insert SECOND_PART at beginning of CURRENT_LINE
         end if
         if current situation is viable for hyphenation
            send WORD to hyphenation process
            copy CURRENT_LINE to PREVIOUS_LINE
         end if
         initialize CURRENT_LINE = ''
      end if

      read WORD from INPUT_FILE

   end loop

   (* last line of input file... *)
   SEND CURRENT_LINE to format/output process

end
```

Figure 10. Pseudocode for the Concurrent Text Formatter

scure the example. This "bug" will, in practice, have little impact on the resulting formatted text.)

The test "if situation is viable for hyphenation" examines such factors as the length of the word to be split and the number of spaces available on the current line in order to determine whether an attempt should be made to split the word, or whether it should be carried to the next line directly.

The main formatting program will require the following declarations for the logical ports:

```
type
   WORDSTRING = packed array[1..MAXWORDSIZE] of char;
   LINESTRING = packed array[1..MAXLINESIZE] of char;

lps
   HYPHENATE : out : (WORDSTRING, byte);
   RESULT    : in  : (WORDSTRING, WORDSTRING);
   FORMAT    : out : (LINESTRING, Boolean);
```

(The second field of the FORMAT l-port is used to indicate to the formatter process when the last line is being sent.)

The communication structures used in the HYPHENATE process and the FORMAT process are straightforward, corresponding to the l-ports in the main process. The communication features in a d-Pascal implementation of the concurrent text formatter are outlined in "Appendix C. Sample Programs" on page 90.

5.3 DATA ABSTRACTION

Consider a very simple form of resource allocation, in which there exists a pool of n identical resources, each with an index number in the range [1..n]. Processes may try to grab a resource from the pool or return a resource to the pool. As each process must have exclusive access to the resource pool, we shall implement the pool as an abstract data structure controlled by a server process.

The server is a very simple process, outlined in Figure 11 on page 61. It can accept two kinds of messages: one to allocate a resource, and one to return a resource to the pool.

```
process SERVER

   initialize representation of resource pool

   loop forever
      if there is at least one resource available then
         wait for either a REQUEST or RELEASE
      else
         wait for a RELEASE only

      if received a REQUEST then
         allocate a resource
      else if received a RELEASE then
         deallocate the released resource
   end loop

end
```

Figure 11. Pseudocode for the Server Process

In the d-Pascal solution, the resource pool will be maintained as a Boolean array R of size MAXPOOLSIZE (a declared constant), the maximum number of resources possible in SERVER's pool. If the i'th resource is available, R[i] will be true; if the i'th resource is not available, R[i] will be false.

```
type
   POOLRANGE = 1..MAXPOOLSIZE;
   POOL = array[POOLRANGE] of Boolean;

var
   R       : POOL;      (* the resource pool                    *)
   N,                   (* actual number of resources in pool   *)
   COUNT,               (* number allocated at any given time   *)
   I       : integer;   (* an index variable                    *)
   RETURN  : byte;      (* the "return address"                 *)
   FOUND   : Boolean;   (* loop control variable                *)
```

Three logical ports are required: one to receive request messages, one to receive release messages, and one to send the index of the allocated resource to a requester. The LPNs of these ports are the declared constants REQUEST, RELEASE and ALLOCATE, respectively. The declarations of the l-ports are:

```
lps
   REQUEST  : in  : (byte);
   RELEASE  : in  : (POOLRANGE);
   ALLOCATE : out : (POOLRANGE);
```

The request message is very simple. Since there is only one type of resource in this example, the type of resource to allocate need not be specified. The process ID of the originator of a request can be found in &SENDER after a WAIT statement senses the presence of the request. The only data required in a request message is the number of the l-port to which the allocate message must be sent (a "return address"). The release message bears the index of the resource to be released, and

the allocate message contains the address of the newly allocated resource.

The resource pool is initialized by simply reading in N, the number of resources, and setting their indicies in R to TRUE. The count of allocated resources will be initialized to 0:

```
read(N);
for I := 1 to N do R[I] := TRUE;

COUNT := 0;
```

The server will loop forever, receiving messages and managing resources. It must ensure that it does not receive a request message if there are no resources available to be allocated, a condition indicated by the value of COUNT being equal to the number of resources:

```
if COUNT = N then
   wait(1port=RELEASE)
else
   wait();
```

If a request is received, the server must search for an available resource, knowing that at least one exists. It must then receive the request message and allocate the resource to the requesting process:

```
(* if request is received: *)
   I := 0;   FOUND := FALSE;
   while (I<N) and not FOUND do begin
      I := I + 1;
      FOUND := R[I];
   end
   receive from &SENDER via REQUEST : (RETURN);
   send to RETURN at &SENDER via ALLOCATE : (I);
   COUNT := COUNT + 1;
   R[I] := FALSE;
```

If a release is received, the resource must be returned to the pool:

```
   receive from &SENDER via RELEASE : (I);
   R[I] := TRUE;
   COUNT := COUNT - 1;
```

A complete d-Pascal program for a server is given in "Appendix C. Sample Programs" on page 90. The example does not try to perform any data verification (for instance, verifying that released resources had actually been allocated to the releaser), although such checks would be implemented in practice.

5.4 QUICKSORT

Quicksort, first presented by C.A.R. Hoare in [HOAR 62], is a recursive sorting algorithm in which disjoint parts of an array are sorted independently. In the single process version of Quicksort, the array is divided into two sections, and the Quicksort procedure is called

recursively on each partition. The recursion terminates when a partition contains one element. The Quicksort algorithm is given in Figure 12 on page 65.

```
procedure QUICKSORT(m,n);
   (* sort partition of array A from A[m] to A[n], inclusive, *)
   (* into non-decreasing order. Record A[m] is arbitrarily  *)
   (* chosen as the pivot record. It is assumed that the     *)
   (* value of the pivot is less than or equal to A[n+1].    *)

   if m < n then begin
      arrange A such that:
        - the pivot is in the correct location in A, call it j
        - all elements of A between A[m] and A[n] which are
             less than the pivot are moved before A[j]
        - all elements of A between A[m] and A[n] which are
             greater than the pivot are moved after A[j]

      call QUICKSORT(m, j-1)     (* recursive call *)
      call QUICKSORT(j+1, n)     (* recursive call *)

end
```

Figure 12. Pseudocode for the main Quicksort process

In the d-Pascal program for Quicksort, the recursive calls can be programmed in one of two ways: in the standard (sequential) fashion, or as the invocation of a concurrent process. Concurrency can be achieved by passing one partition to another process while sorting the second partition.

The main process must be responsible for reading the initial array from disk, and writing the sorted array back to disk. All other processes to which partitions were sent during the recursion input their data from and output their results to l-ports. All processes require

the following declarations, assuming a declared constant MAXSIZE, the largest possible data array that can be sorted:

```
type
   VECTOR = array[1..MAXSIZE] of integer;

var
   A   : VECTOR;     (* the data array                *)
   M,N : integer;    (* range of partition to be sorted *)
```

Each process requires five l-ports: one l-port for sending and receiving unsorted data along with the bounds of the partition to be sorted, one for sending and receiving the sorted array, and three to communicate with the resource server:

```
lps
   UNSORTED : in,out : (VECTOR, integer, integer);
   SORTED   : in,out : (VECTOR);
   REQUEST  : out    : (byte);
   RELEASE  : out    : (byte);
   RESOURCE : in     : (byte);
```

The resource server is required because it is not possible to create processes dynamically in d-Pascal. A pool of Quicksort worker processes must be created when the d-job is initiated, and they are used (invoked) when their process ID is chosen by the server. The server must be initiated as part of the d-job.

The main process must input the data, sort it, and then output the sorted data. (Assume the data is simply a sequence of integers.)

```
begin    (* Quicksort main *)

   read(SIZE);
   for I := 1 to SIZE do read(A[I]);
   A[SIZE+1] := MAXINT;

   call QUICKSORT(1,SIZE);

   for I := 1 to SIZE do write(A[I]);

end.    (* Quicksort main *)
```

The worker processes are similar to the main process, except that they receive and send instead of read and write:

```
begin    (* Quicksort worker *)

   wait(lport=UNSORTED);
   receive from &SENDER via UNSORTED : (A | M | N);

   call QUICKSORT(M,N);

   send to SORTED at &SENDER via SORTED : (A);

end.    (* Quicksort worker *)
```

The quicksort procedure is the same in the main process and in the workers. If the partition contains at least one member, it partitions the partition, using the M'th element as the pivot:

```
procedure QUICKSORT(M,N : integer);

var
   I,J,PIVOT,TEMP : integer;

begin
   if M < N then begin
      I := M;  J := N+1;  PIVOT := A[M];
      repeat
         repeat I := I+1 until A[I] >= PIVOT;
         repeat J := J-1 until A[J] <= PIVOT;
         if I<J then begin   (* swap A[I] with A[J] *)
            TEMP := A[I];  A[I] := A[J];  A[J] := TEMP;
         end
      until I >= J;
      TEMP := A[M];  A[M] := A[J];  A[J] := TEMP;
```

At this point, the pivot is at the J'th position of A, and the following hold:

A[i] <= A[J], for all i <= J;

A[i] >= A[J], for all i >= J;

Now, the partitions must be sorted. The first partition will be sent to another process to be sorted concurrently, while the other partition will be sorted by the current process. The program will ask the server process for the process ID of an available worker process. Note that it sends the LPN of the RESOURCE 1-port to the server, so that the server will know to which 1-port of the quicksort process to send its response message.

```
send to SERVERREQ at SERVER via REQUEST : (RESOURCE);
receive from SERVER via RESOURCE : (PROCESSID);
send to UNSORTED at PROCESSID via UNSORTED : (A | M | J-1);

call QUICKSORT(J+1, N);
```

When the call to QUICKSORT returns, A is sorted from A[J] to A[N], inclusive. The elements from A[M] to A[J-1] must be received from the concurrent process into a temporary array and copied into A. (If the RECEIVE statement named A directly, the part that is already sorted will get overwritten.) A is then sent back to the process which sent it to the current one, and the worker process which just completed the sorting can be returned to the resource pool:

```
receive from PROCESSID via SORTED : (B);
for I := M to J-1 do A[I] := B[I];
send to SORTED at &SENDER via SORTED : (A);
```

The complete code for QUICKSORT is given in "Appendix C. Sample Programs" on page 90. There are several ways of improving the performance of the d-job. One way would be to allow a process to sort partitions internally if the were not "large enough" to merit being sent over the network to another process. Another way would be to use a slightly more complex server, which would return a negative response if no resources were available. This also would allow a process to decide to sort a partition internally. (The d-job presented here would block while waiting for a resource to be allocated to it.)

Chapter 6

SUMMARY

This thesis presents d-Pascal, a programming language for a loosely-coupled network of computers, which successfully implements layer seven of the ISO-OSI Reference model [TANE 81]. d-Pascal is an asynchronous message passing language which augments standard Pascal with a set of constructs which permit interprocess communication.

Each process in d-Pascal is called a **module**, and is a separately compiled program with its own private address space, and its own unique **process ID**. Each module may declare **logical ports**, or l-ports, which act as windows through which messages can be sent or received. Each l-port has a type structure, and the messages which pass through them should be of comformable type.

The SEND and RECEIVE statements are used for message communication. The SEND statement is non-blocking: a sending process may send messages at any time, irrespective of whether the intended destination processes are ready to receive them. The SEND statement must name the process ID of the intended receiver, the local l-port through which the message is to be sent, and the external l-port through which the receiver is to receive the message, as well as provide the data, in the form of expressions, which are to be sent in the body of the message. The RECEIVE statement must name the process ID of the originator of the message, the local l-port to which the message was sent, and the variables which are to be assigned the contents of the fields of the message. The process executing a RECEIVE statement will block if no message of the kind re-

quested has been sent to it. When such a message becomes available, the data will be assigned to the specified variables and the RECEIVE statement will terminate. Language facilities also exist to determine whether a message from a particular process, and/or sent to a particular local l-port, is available to be received.

That no explicit data structure for messages is required in either d-Pascal or its preprocessor may sound surprising, but the actual construction of messages from d-Pascal expressions, and the splitting of messages into their component fields to be received, are ISO-OSI layer five functions, and are beyond the scope of the application language. Of course, the implementation of the language must provide an interface between the compiled program and the underlying message handling software of the operating system, while providing the programmer with a direct virtual interface between processes. In d-Pascal, this interface between processes takes the form of the logical ports in the origin and destination processes.

d-Pascal has been implemented as a preprocessor for sequential Pascal on Cuenet [GROS 82, RADH 85], a distributed network at Concordia University that is described in "Appendix A. The Cuenet Facility for Distributed Computing". The closeness of d-Pascal to widely known standard Pascal makes it a fairly quick language to learn. The actual line-by-line coding of d-Pascal is very similar to coding in sequential Pascal: the only dramatically new syntax appears in the LPS declaration section, and the SEND and RECEIVE statements.

One difficulty with the d-Pascal environment became evident while writing d-Pascal programs: the process IDs of all the modules must be

known at compile time. This makes it difficult to create a module which is intended to be used in more than one d-job, such as a utility like the SERVER process discussed in Chapter Five, because every module which wants to communicate with it must know its process ID. (The SERVER did not have to know at compile time the process IDs of the modules which used it, because the process IDs of the processes which send messages to it are found in &SENDER each time SERVER received one of their messages.) The situation could be rectified if, at the time the d-job is loaded onto the network, each SOS had access to a table relating names of processes to their process IDs. Applications would then refer to other processes by name, and the message handling software of the SOS would convert the names to process IDs at run time. (Solutions to the naming problem in a distributed system have been reported in the literature [TANE 85]. They could be incorporated in future extensions of this work.) The quicksort example, which uses the SERVER process, would be much simpler if the operating system provided the means for duplicating processes, relieving the programmer of the burden of resource management. (The quicksort application is more suited to a language such as Pascal-C [LAM 82], in which new instances of down procedures are created as required.)

The d-Pascal preprocessor is now fully operational. Operating system support on Cuenet includes the implementation of the send and receive operations, including the queuing of messages. One important feature still outstanding is the integration of the reconfigurability aspect of Cuenet into the d-Pascal d-jobs, to allow programmers to experiment with algorithms on different network configurations.

REFERENCES

[ADA 81] --. *The Programming Language Ada, Reference Manual.* Proposed Standard Document, United States Department of Defense, Lecture Notes in Computer Science, 106, Springer-Verlag, 1981.

[ANDR 78] E. Andre, P. Decitre. *On Providing Distributed Application Programmers with Control over Synchronization.* Computer Networks Protocols Symposium, Liege, Belgium, 1978.

[BIRR 84] A. D. Birrell, B. J. Nelson. *Implementing Remote Procedure Calls.* ACM Transactions on Computer Systems, vol. 2, no. 1, February, 1984, 39-59.

[BRIN 73] P. Brinch Hansen. *Operating System Principles.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1973.

[BRIN 75] P. Brinch Hansen. *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering, vol. SE-1, no. 2, June, 1975.

[BRIN 77] P. Brinch Hansen. *The Architecture of Concurrent Programs.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1977.

[BRIN 78a] P. Brinch Hansen, J. Staunstrup. *Specification and Implementation of Mutual Exclusion.* IEEE Transactions on Software Engineering, vol. SE-4, 1978, 365-370.

[BRIN 78b] P. Brinch Hansen. *Distributed Processes: A Concurrent Programming Structuring Concept.* Comm. ACM 21, November, 1978, 934-941.

[BRIN 81a] P. Brinch Hansen. *Edison--a Multiprocessor Language.* Software -- Practice and Experience, vol. 11, 325-361, 1981.

[BRIN 81b] P. Brinch Hansen. *The Design of Edison.* Software -- Practice and Experience, vol. 11, 1981, 363-396.

[CIVE 82] P. Civera, G. Conte, D. Del Cors , F. Gregoretti, E. Pasero. *The µ* Project: An Experience with a Multimicroprocessor System.* IEEE Micro, vol. 2, no. 2, May, 1982, 38-49.

[CORD 81] J. R. Cordy, R. C. Holt. *The Specification of Concurrent Euclid.* Technical Report CSRG-133, Computer Systems Research Group, University of Toronto, August, 1981.

[DIJK 68] E. W. Dijkstra. *Co-operating Sequential Processes.* Programming Languages, Academic Press, New York, 1968, pp. 43-112.

[DIJK 75] E. W. Dijkstra. *Guarded Commands, Nondeterminacy, and Formal Derivation of Programs.* Comm. ACM 18, August, 1975, 453-457.

[FATH 83] E. T. Fathi, M. Krieger. *Multiple Microprocessor Systems: What, Why, and When.* IEEE Computer, March, 1983, 23-32.

[FELD 79] J. A. Feldman. *High Level Programming for Distributed Computing.* Comm. ACM 22, June, 1979, 353-368.

[GIBB 81] D. R. Gibby, N. W. Bennee. *P-6800 User Guide and Reference Manual.* PO Box 128, Cambridge, CB2 5EZ, England, 1981.

[GROS 82] C. Grossner. *The Development and Implementation of CUENET: a Reconfigurable Network of Loosely Coupled Microcomputers.* Master's Thesis, Concordia University, Montreal, Canada, 1982.

[HOAR 62] C. A. R. Hoare. *Quicksort.* Computer Journal, vol. 5, no. 1, 1962, 10-15.

[HOAR 72] C. A. R. Hoare. *Towards a Theory of Parallel Programming.* In Operating System Techniques, Academic Press, New York, 1972.

[HOAR 74] C. A. R. Hoare. *Monitors: An Operating System Structuring Concept.* Comm. ACM 17, October, 1974, 549-557.

[HOAR 78] C. A. R. Hoare. *Communicating Sequential Processes.* Comm. ACM 21, August, 1978, 666-677.

[HOLT 83] R. Holt. *Concurrent Euclid, the UNIX System, and TUNIS.* Addison-Wesley Publishing Co. Inc., 1983.

[JENS 74] K. Jensen, N. Wirth. *Pascal User Manual and Report.* Springer-Verlag, 1974.

[JOSE 81] M. Joseph. *Communication Schemes.* Computer Science Department, Carnegie-Mellon University, Pittsburgh, Pa., April, 1981.

[LAM 82] C. Lam, J. W. Atwood, S. Cabilio, B. C. Desai, P. Grogono, J. Opatrny. *A Multiprocessor Project for Combinatorial Computing.* CIPS 1982 National Conference, Saskatoon, Saskatchewan, May, 1982, 325-329.

[LAMP 77] B. Lampson, J. Horning, R. London, J. Mitchell, G. Popek. *Report on the Programming Language Euclid.* SIGPLAN Notices 12, 1, February, 1977.

[LAUE 75] S. Lauesen. *A large semaphore based operating system.* Comm. ACM 18, July, 1975, 377-389.

[LINT 81] B. Lint, T. Agerwala. *Communications Issues in the Design and Analysis of Parallel Algorithms.* IEEE Transactions on Software Engineering, vol. SE-7, March, 1981, 174-188.

[LORI 72] H. Lorin. *Parallelism in Hardware and Software: Real and Apparent Concurrency*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1972.

[MILN 80] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, 92, Springer Verlag, 1980.

[RADH 85] T. Radhakrishnan, C. Grossner. *Cuenet -- A Distributed Computing Facility*. IEEE Micro, vol. 5, no. 1, February, 1985, 42-52.

[SILB 79] A. Silberschatz. *Communication and Synchronization in Distributed Systems*. IEEE Transactions on Software Engineering, vol. SE-5, November, 1979, 542-546.

[STAU 76] J. Staunstrup, S. Sorensen. *Platon: A High Level Language for Systems Programming*. in Minicomputer Software, North Holland, 1976.

[STAU 82] J. Staunstrup. *Message Passing Communication versus Procedure Call Communication*. Software -- Practice and Experience, vol. 12, 1982, 223-234.

[SWAN 77] R. J. Swan, S. H. Fuller, D. P. Siewiorek. *Cm* -- A Modular Microprocessor*. AFIPS Conference Proceedings, 1977 NCC, 637-644.

[TANE 81] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall Software Series, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1981.

[TANE 85] A. S. Tanenbaum, R. van Renesse. *Distributed Operating Systems*. ACM Computing Surveys, vol. 17, no. 4, December, 1985.

[WIRT 71] N. Wirth. *The Programming Language Pascal*. Acta Informatica 1, 1, 1971, 35-63.

[WIRT 77] N. Wirth. *Modula: A Language for Modular Programming*. Software -- Practice and Experience, vol. 7, January, 1977, 3-35.

[WITT 78] L. D. Wittie. *MICRONET: A Reconfigurable Microcomputer Network for Distributed Systems Research*. Simulation, vol. 31, no. 5, November, 1978, 145-153

Appendix A

THE CUENET FACILITY FOR DISTRIBUTED COMPUTING

The diversity of multicomputer systems is as great as the diversity of languages which have been developed to program them. This Appendix presents a brief overview of multicomputer systems, followed by a more detailed description of Cuenet, the facility upon which much of the work of this thesis is implemented.

A.1 DISTRIBUTED COMPUTING SYSTEMS

A distributed computing system consists of several interconnected computers. The computers are generally under the control of a single operating system, although each computer may also have its own local operating system. One rationale for building a distributed system is that several computers working concurrently may be able to solve a problem faster than a single computer. If diverse computations are required, each processing site may be specially designed to carry out a particular type of task. Alternatively, if reliability is a prime consideration, a distributed computing system may be designed to introduce redundancy, by having all processors perform the same operations and then compare results. This concept is used in the shuttle orbiters of NASA's Space Transportation System (the "Space Shuttle" program in the United States). Each shuttle orbiter contains four identical, synchronized computers running identical software.

Two properties which can be used to classify multicomputer systems are the degree of coupling that exists between processing sites, and the topology in which the processing sites are arranged [FATH 83]. The degree of coupling exhibited in a multicomputer system can range from tight coupling to loose coupling. In a tightly coupled system, all computers share a common memory address space, and communicate through this shared memory. In a loosely coupled system, each computer has its own private memory address space, and communicates with other computers by passing messages over a medium linking the computers. The pattern in which computers are linked to each other is called the **topology** of the multicomputer. Some common topologies are shown in Figure 13 on page 78.

Examples of research projects in distributed computing include:

1. Cm* [SWAN 77] at Carnegie-Mellon University. Cm* is a group of tightly coupled LSI-11 microcomputers. The system's topology is a fixed, two-level hierarchy.

2. Micronet [WITT 78] at the State University of New York. Micronet is a loosely coupled network of microcomputers connected in a matrix topology.

3. μ* [CIVE 82] at Politecnico di Torino, Italy. μ* uses a multi-level star topology, in which clusters of processors are connected over intercluster links.

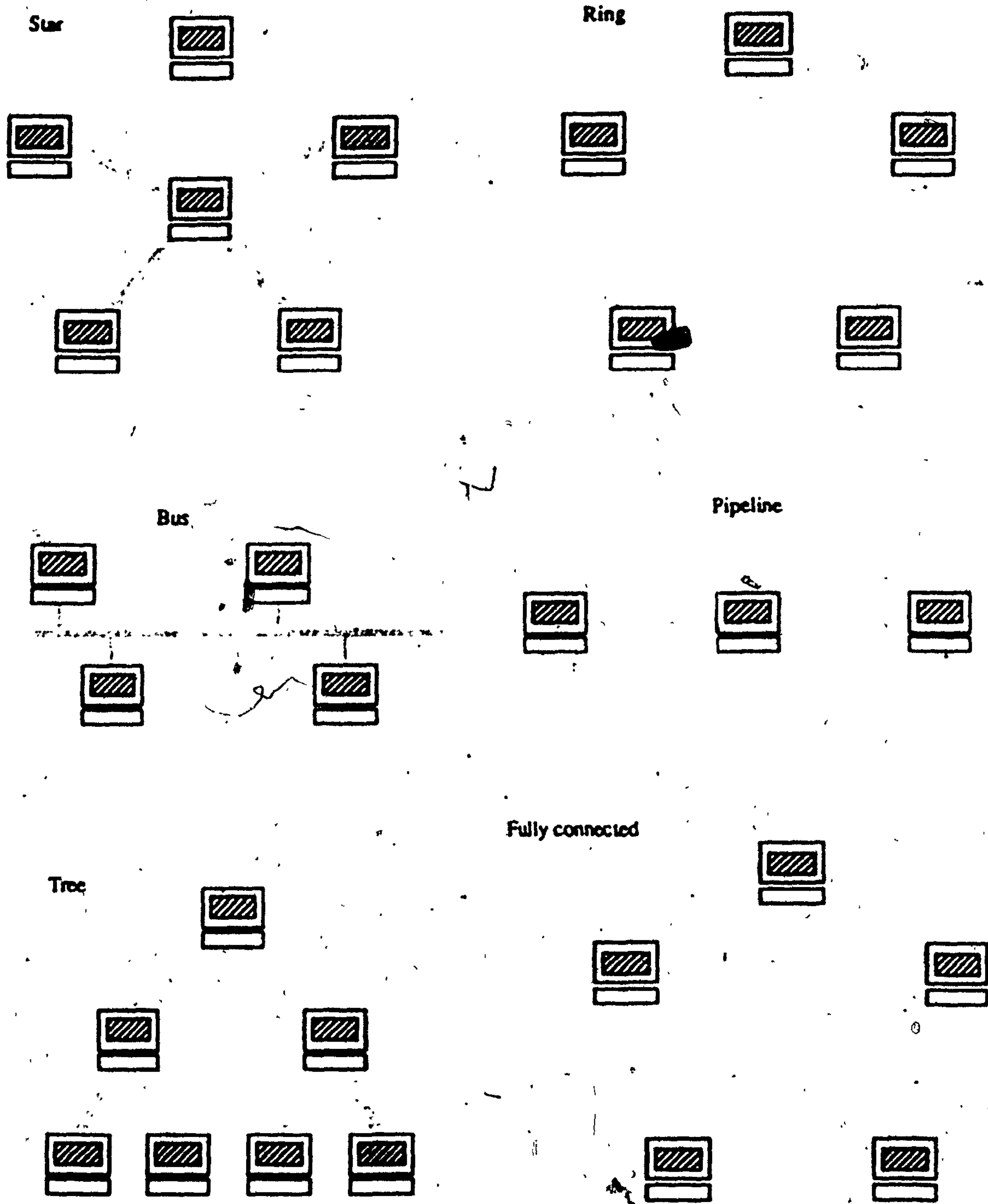

Figure 13. Multicomputer Topologies: star, ring, bus, pipeline, tree and fully connected topologies.

4. Cuenet [GROS 82, RADH 85] at Concordia University. Cuenet is a loosely coupled network of heterogenous computers, capable of being configured into any topology desired by the applications programmer.

A.2 CUENET

Cuenet is a reconfigurable network of microcomputers which uses one or more C-buses as the interconnection medium. Each computer is connected to C-bus through a dedicated C-bus interface. Figure 14 on page 80 shows the configuration of Cuenet.

Cuenet is said to be reconfigurable because it is capable of assuming whatever topology is required by the application. Whereas Figure 14 shows that all computers are physically interconnected via C-buses, each C-bus interface contains an access vector which defines the set of other computers with which each computer can communicate. By defining appropriate access patterns, the application programmer can logically create any topology. Access vectors are discussed in "Programmer-specified Configurability" on page 83.

A.2.1 The Components of Cuenet

The components of Cuenet are described in this section.

The Master Computer: It is through the master computer that the applications programmer interfaces with Cuenet. The master is responsible for the following operations:

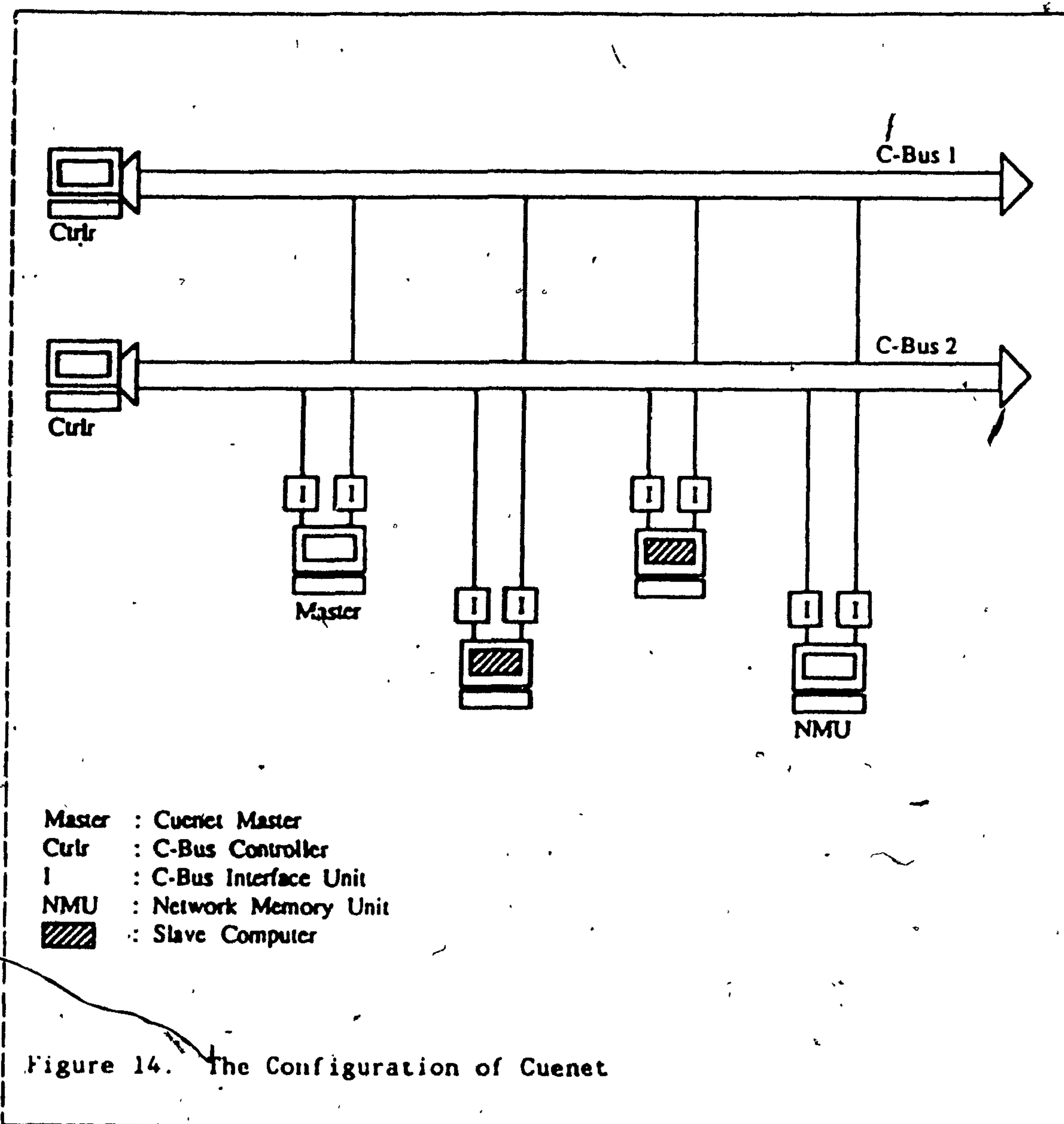

Figure 14. The Configuration of Cuenet

- the allocation of slaves to an application program;
- loading of the modules resulting from a program decomposition onto the appropriate slaves;
- initializing the access vectors to configure Cuenet into the topology required by the application;

- processing exception conditions, including program termination, irrecoverable errors and access rights violations.

Cuenet contains only one master, but since it does not contain any special hardware, any computer in Cuenet that can run the master software can function as the master. One computer must be acting as the master at all times. If the master fails, distributed computing on Cuenet is halted until the master function is reinstated on another computer.

Note that application programs never run on the Cuenet master.

The Slave Computers: There can be up to 63 slave computers in Cuenet. The slaves execute the modules of a program decomposition.

Cuenet may be a heterogeneous network in that the slaves do not have to all be of the same type. A slave may be a special purpose computer, such as a processor for signal processing. The slave's native operating system must be augmented to handle the responsibilities of Cuenet. The augmented operating system is called a **slave operating system** (SOS).

Slaves may be either 8-bit or 16-bit computers, and may have whichever peripherals they require.

C-bus: Communication between computers on Cuenet is via messages transmitted over one or more timeshared buses called C-buses. The C-bus interconnection mechanism was designed to be fast enough to support message traffic generated by a distributed algorithm, yet flexible

enough to be used as an experimental research tool. Physically, C-bus consists of twisted pairs of wires logically grouped into four categories: data lines, send address lines, receive address lines and control lines. Computers are plugged into T-junctions inserted at arbitrary locations along the bus.

The C-bus Controller: Each C-bus is managed by a C-bus controller. This centralized controller consolidates the hardware required for bus arbitration and control into a single unit, thus reducing the complexity required at each local station. Centralization comes at the cost of reliability, however. If the reliability of Cuenet is critical, more than one C-bus should be used.

C-bus Interface Units: Each computer is connected to C-bus through a C-bus interface unit. Each unit includes input and output buffers and queues for messages, an access vector, lock and key registers, and status and control registers.[6] It is due to the interface units that the integrity of the user defined topology is maintained.

The Network Memory Unit: A common storage facility is available on Cuenet. This function is provided by specialized slave computers called NMU's, or Network Memory Units. Each NMU is a general pur-

---

[6] The functions of these elements are described in "Programmer-specified Configurability" on page 83 and "How Messages are Transmitted" on page 84.

pose microcomputer with a large memory. NMU's are considered to be shared resources by the other slaves. Centralizing the storage and maintenance of common data in intelligent NMU's saves memory by avoiding duplication of data, and saves time by reducing the number of messages transmitted over C-bus whenever the common data needs to be updated.

A.2.2 Programmer-specified Configurability

It is possible to configure the processing sites in Cuenet into any topology desired by the application programmer. This is accomplished by restricting the communication patterns between computers, which are all physically interconnected by C-bus.

Each computer in Cuenet is associated with a C-bus interface unit, and each interface unit contains an **access vector**, which is essentially a list of all other Cuenet computers with which a given computer is permitted to communicate. The access vectors can only be written to by the Cuenet master.

C-bus interface units also contain **lock** and **key registers**. A set of mechanical switches is used to set the lock registers on each interface unit to a value called the **combination**. A program can only access the interface by unlocking it, that is, by writing the combination into the key register. In a "friendly" environment, it can be assumed that the system software knows the combination, but the application software does not: therefore, an application program can only access the interface through calls to the system software (slave operating system).

A.2.3 How Messages are Transmitted

Each slave operating system on Cuenet includes message communication software containing the data structures shown in Figure 15 on page 85.

When an application program wants to send a message, it supplies the SOS with the message data and the intended destination of the message. The SOS builds the message header and footer (see Figure 16 on page 86), places the message in the interface's send buffer, and asserts one of the C-bus control lines, the Bus Request line. (If the send buffer, which has space for one message only, is full, the message is placed in the send queue.) Control is then returned to the application module, which will now execute as if the message has been sent.

When the C-bus controller senses the presence of the message in the send buffer, it interrupts the slave application, places the message onto C-bus, and directs it to the destination specified in the message header. When the send buffer is free, the slave's message handling software will move the next message from the head of the send queue (if one is there) to the send buffer, and control will return to the application.

When a message is placed in the send buffer, a copy is written to the message backlog, in case retransmission is requested.

The C-bus controller will transfer the message to the receive buffer of the destination slave, and cause an interrupt in the receiver's message handling software. The message will then be transferred to

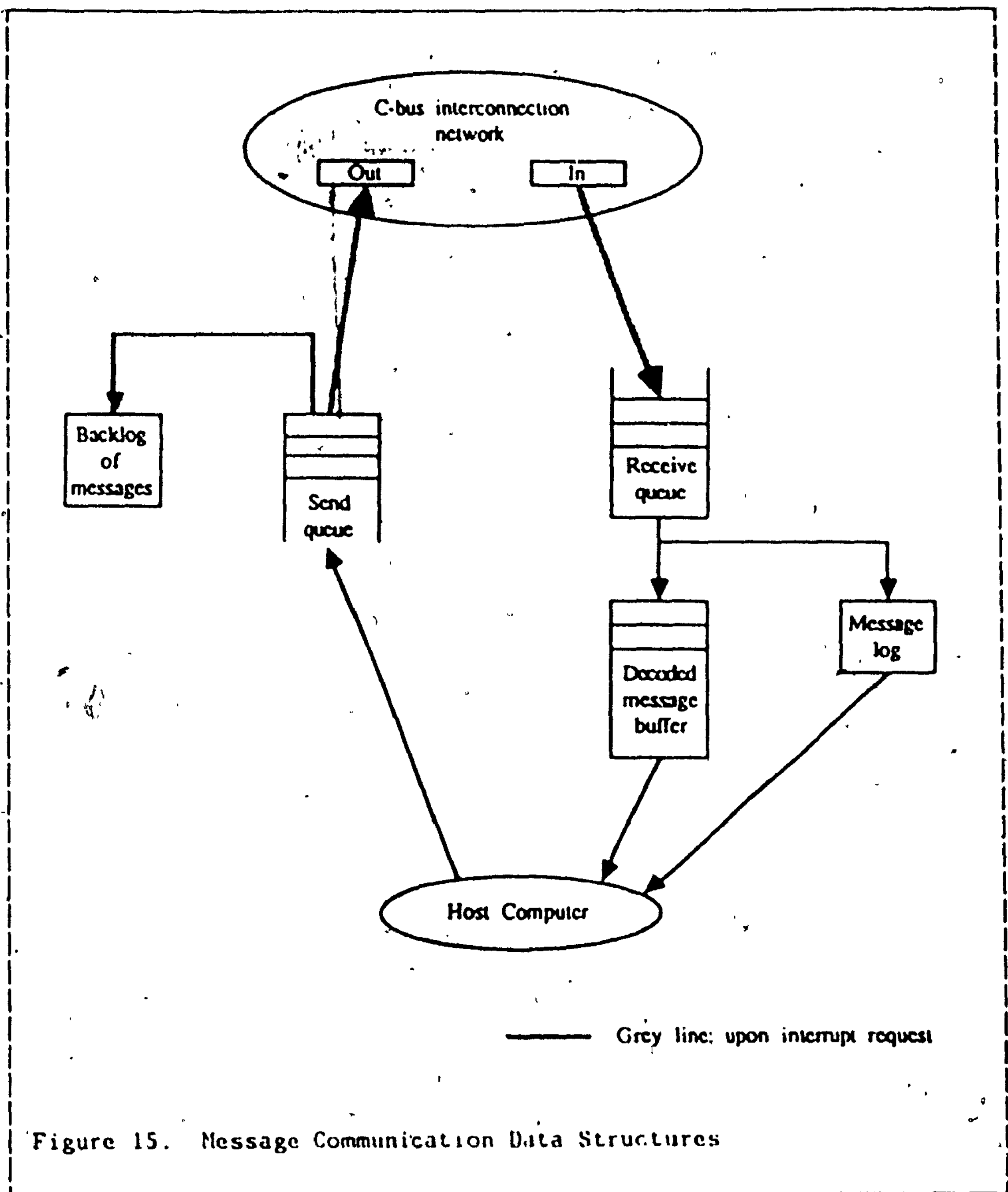

Figure 15. Message Communication Data Structures

the receive queue for processing by the SOS, which decodes the header, transfers the data portion into the decoded message buffer, and makes an entry in the message log.

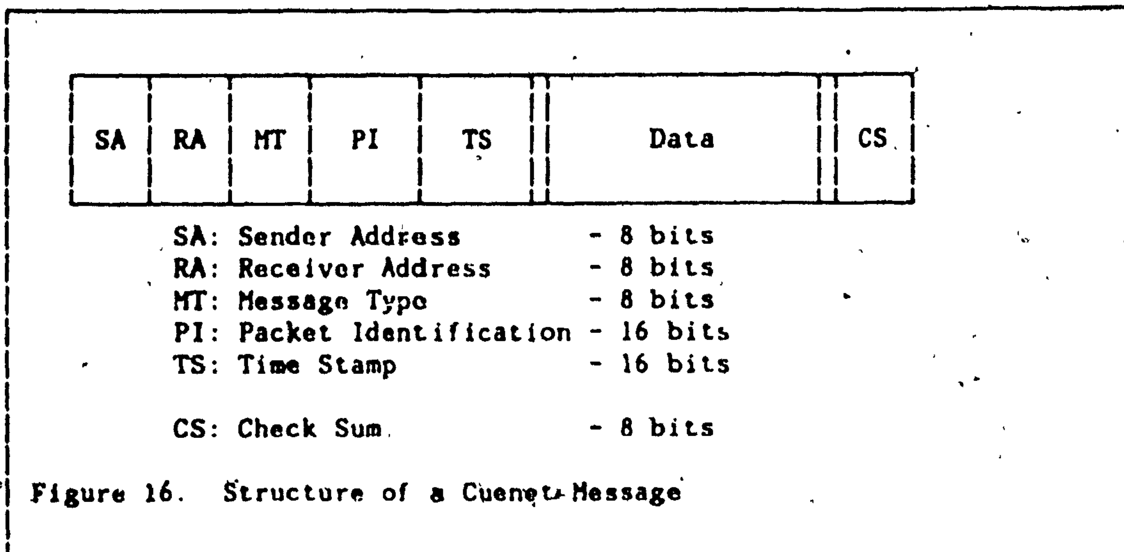

Figure 16. Structure of a Cuenet Message

The application program receives the message by searching for its entry in the message log and retrieving it from the decoded message buffer. The application is not signaled when a message is received by the interface unit: it must query the log to determine if a particular message is present.

It is important to note that "sending" and "receiving" have different connotations in different contexts. An application program views sending and receiving as atomic actions, totally abstracted from the physical aspects of message transmission seen from the SOS perspective. The application considers a message to be sent as soon as it is passed off to the local SOS: it is unaware of when the message actually leaves the C-bus interface, as SOS message handling software is interrupt driven and totally transparent to the application. Similarly, a slave is unaware of the physical reception of a message by its SOS. A message can only be logically received by an application after it checks the

message log, which is updated by the SOS when to message is physically received.

Most significantly, a sending application is unaware of when the message is received by the destination application, and the reception of a message has no effect on the originator. In fact, the sending application is not even aware of exactly when its SOS physically sends the message across C-bus.

A.2.4 The Structure of a Cuenet Message

Figure 16 on page 86 shows the structure of a Cuenet message. A message is divided into three parts:

1. The Header

- Sender Address - the physical slave number of the sender. The network must have a unique physical address for each slave.
- Receiver Address - the physical slave number of the intended receiver of the message.
- Message Type - a message may be either a standard application message (between slaves), a command message sent from the master to a slave operating system, or a program or file segment being sent between the master and a slave.
- Packet Identification - used to identify messages, to request resends, to reconstruct logical messages which required more than one physical message for their transmission, etc.

- Time Stamp - the time that the message was sent, to maintain the correct temporal sequencing.

2. The Data Portion:

   The data being sent by the application. Cuenet does not interpret this data in any way.

3. The Footer:

   An eight bit check sum. If the check sum does not verify, the receiving message handling software will request a retransmission of the message.

The length of a physical message is 256 bytes. The data portion is 248 bytes long.

Appendix B

SYNTAX OF d-PASCAL EXTENSIONS

This Appendix summarizes the extensions to standard Pascal required by d-Pascal.

```
<LPSDECL>      ::= LPS <LPSDEF> { ; <LPSDEF> } ; | empty
<LPSDEF>       ::= <LPS CONST> : <LPSMODE> : <LPSLIST>
<LPSMODE>      ::= IN | OUT | IN,OUT | OUT,IN
<LPSLIST>      ::= ( <ITEMS> )
<ITEMS>        ::= empty | <type ID> { , <type ID> }
<LPS CONST>    ::= <numeric one byte global CONST id or
                                                literal constant>

<SEND>         ::= SEND TO <COMMARG> AT <COMMARG> VIA <LPS CONST> :
                                                          <SENDLIST>
<SENDLIST>     ::= ( <EXPRESSION> { <BAR> <EXPRESSION> } )
<RECEIVE>      ::= RECEIVE FROM <COMMARG> VIA <LPS CONST> : <RCVLIST>
<RCVLIST>      ::= ( <ID> { <BAR> <ID> } ) | ( )
<COMMARG>      ::= <var ID> | <SYSVAR> | <LPS CONST>
<BAR>          ::= <the vertical bar character "|">

<MESSAGE>      ::= MESSAGE ( <ARGLIST> )
<WAIT>         ::= WAIT ( <ARGLIST> )
<ARGLIST>      ::= empty | <LPORT ARG> | <LPORT ARG> , <SENDER ARG> |
                     <SENDER ARG> | <SENDER ARG> , <LPORT ARG>
<LPORT ARG>    ::= LPORT = <byte-valued expression>
<SENDER ARG>   ::= SENDER = <byte-valued expression>

<SYSVAR>       ::= & <SYSVARNAME>
<SYSVARNAME>   ::= SENDER | LPORT
```

Appendix C

APPENDIX C. SAMPLE PROGRAMS

C.1 THE MERGING PROGRAM

```
program MERGE;

   const
      STREAM = 1;
      P1 = 10;  (* process ID of P1 *)
      P2 = 11;  (* process ID of P2 *)
      C  = 12;  (* process ID of C  *)

   lps
      STREAM : in,out : (integer);

   var
      V1, V2 : integer;   (* integer from P1, P2, respectively *)
      DONE   : Boolean;

   begin   (* MERGE *)
      receive from P1 via STREAM : (V1);
      receive from P2 via STREAM : (V2);

      DONE := FALSE;
      repeat
         if V1<V2 then begin
            send to 1 at C via STREAM : (V1);
            receive from P1 via STREAM : (V1);
         end
         else if V2<V1 then begin
            send to 1 at C via STREAM : (V2);
            receive from P2 via STREAM : (V2);
         end
         else if V1<>MAXINT then begin
            send to 1 at C via STREAM : (V1);
            receive from P1 via STREAM : (V1);
            receive from P2 via STREAM : (V2);
         end
         else   (* V1=V2=MAXINT *)
            DONE := TRUE;
      until DONE;

      send to 1 at C via STREAM : (MAXINT);  (* end of merged list *)
   end.    (* MERGE *)
```

C.2 THE TEXT FORMATTER PROGRAMS

```
program CONCURRENT_FORMATTER;

   const
      HYPHENATE = 1;               (* local LPN *)
      RESULT    = 2;               (* local LPN *)
      FORMAT    = 3;               (* local LPN *)
      HYPH_PROCESS = 2;            (* external process ID *)
      FORM_PROCESS = 3;            (* external process ID *)
      HYPH_LPORT   = 1;            (* external LPN *)
      FORM_LPORT   = 1;            (* external LPN *)
      MAXWORDSIZE = 32;
      MAXLINESIZE = 80;

   type
      WORDSTRING = packed array[1..MAXWORDSIZE] of char;
      LINESTRING = packed array[1..MAXLINESIZE] of char;

   lps
      HYPHENATE : out : (WORDSTRING, byte);
      RESULT    : in  : (WORDSTRING, WORDSTRING);
      FORMAT    : out : (LINESTRING);

   var
      WORD,
      FIRST_PART,
      SECOND_PART   : WORDSTRING;
      CURRENT_LINE,
      PREVIOUS_LINE : LINESTRING;
      HYPH_PENDING  : Boolean;  (* TRUE when a call to the *)
                                hyphenation process is pending *)
      N_LEFT        : byte;     (* number of character positions
                                remaining in current line      *)

   begin   (* CONCURRENT_FORMATTER *)
      (* INITIALIZATION: - set CURRENT_LINE to all blanks
                         - set HYPH_PENDING to FALSE
                         - set all formatting parameters
                           and line indicies
                         - read the first word into WORD *)

      while not eof(input) do begin
         if (* WORD fits on CURRENT_LINE *) then begin
            (* add WORD to CURRENT_LINE *)
            N_LEFT := (* no. of character positions remaining *)
         end
         else begin
            if HYPH_PENDING then begin
               receive from HYPH_PROCESS via RESULT :
                                       (FIRST_PART | SECOND_PART);
               (* add FIRST_PART to end of PREVIOUS_LINE *)
               send to FORM_LPORT at FORM_PROCESS via FORMAT :
                                       (PREVIOUS_LINE | FALSE);
```

```
            (* insert SECOND_PART at beginning of CURRENT_LINE *)
         end
         if (* hyphenation attempt is viable *) then begin
            send to HYPH_LPORT at HYPH_PROCESS via HYPHENATE :
                                        (WORD | N_LEFT);
            PREVIOUS_LINE := CURRENT_LINE;
            HYPH_PENDING := TRUE;
         end
         else
            HYPH_PENDING := FALSE;
      end;

      (* read next WORD from input *)

   end;   (* while not eof... *)

   (* send last line, indicate end of file *)
   send to FORM_LPORT at FORM_PROCESS via FORMAT :
                                 (CURRENT_LINE | TRUE);

end.   (* CONCURRENT_FORMATTER *)
```

```
program HYPHENATE;

   const
      SPLIT_THIS   = 1;      (* local LPN *)
      RESULT       = 2;      (* local LPN *)
      RESULT_LPORT = 2;      (* external LPN *)
      MAXWORDSIZE  = 32;

   type
      WORDSTRING = packed array[1..MAXWORDSIZE] of char;

   lps
      SPLIT_THIS : in   : (WORDSTRING,byte);
      RESULT     : out  : (WORDSTRING, WORDSTRING);

   var
      WORD,
      FIRST_PART,
      LAST_PART  : WORDSTRING;
      MAX_FIRST  : byte;        (* maximum size permitted for
                                   FIRST_PART                    *)

   begin   (* HYPHENATE *)
      while TRUE do begin;
         wait(lport=SPLIT_THIS);
         receive from &SENDER via SPLIT_THIS : (WORD | MAX_FIRST);

         (* attempt to hyphenate WORD, such that first part has
            fewer than MAX_FIRST letters.  If successful, assign
            the first part of the split word to FIRST_PART (add
            a hyphen to the end of the string), and the second
            part of the split word to SECOND_PART.  If not suc-
            cessful, assign blanks to FIRST_PART and WORD to
            SECOND_PART.                                          *)

         send to RESULT_LPORT at &SENDER via RESULT :
                                     (FIRST_PART | SECOND_PART);
      end   (* while forever... *)

   end.  (* HYPHENATE *)
```

```
program FORMAT_LINE;

   const
      MAXLINESIZE = 80;

   type
      LINESTRING = packed array[1..MAXLINESIZE] of char;

   lps
      1 : in : (LINESTRING,Boolean);

   var
      TEXTLINE : LINESTRING;
      EOF_FLAG : Boolean;     (* true on message containing
                                             last line *)

   begin   (* FORMAT_LINE *)
      wait(lport=1);

      repeat
         receive from &SENDER via 1 : (TEXTLINE | EOF_FLAG);
         (* format TEXTLINE in suitable form for
            output device                               *)
         (* output formatted line to output device *)
      until EOF_FLAG;

   end.   (* FORMAT_LINE *)
```

C.3 THE SERVER PROGRAM

```
program SERVER;

   const
      MAXPOOLSIZE = 500;
      REQUEST     = 1;    (* logical port numbers *)
      RELEASE     = 2;
      ALLOCATE    = 3;

   type
      POOLRANGE   = 1..MAXPOOLSIZE;
      POOL        = array[POOLRANGE] of Boolean;

   lps
      REQUEST  : in  : (byte);
      RELEASE  : in  : (POOLRANGE);
      ALLOCATE : out : (POOLRANGE);

   var
      R      : POOL;      (* the resource pool                *)
      N,                  (* actual number of resources in pool *)
      COUNT,              (* number allocated at any given time *)
      I      : integer;   (* an index variable                *)
      RETURN : byte;      (* the "return address"             *)
      FOUND  : Boolean;   (* loop control variable            *)

   begin   (* SERVER *)

      read(N);
      for I := 1 to N do R[I] := TRUE;
      COUNT := 0;

      repeat   (* loop forever *)

         if COUNT = N then
            wait(lport=RELEASE)
         else
            wait();

         if &LPORT=REQUEST then begin   (* a request is received *)
            I := 0;  FOUND := FALSE;
            while (I<N) and not FOUND do begin
               I := I + 1;
               FOUND := R[I];
            end
            receive from &SENDER via REQUEST : (RETURN);
            send to RETURN at &SENDER via ALLOCATE : (I);
            COUNT := COUNT + 1;
            R[I] := FALSE;
         end
         else begin   (* a release is received *)
            receive from &SENDER via RELEASE : (I);
            R[I] := TRUE;
```

```
            COUNT := COUNT - 1;
        end

    until FALSE   (* end of loop forever *)

end.    (* SERVER *)
```

C.4 THE QUICKSORT PROGRAMS

```
program QUICKSORTMAIN;

   const
      MAXSIZE  = 1000;
      UNSORTED = 1;     (* logical port numbers *)
      SORTED   = 2;
      REQUEST  = 3;
      RELEASE  = 4;
      RESOURCE = 5;

   type
      VECTOR = array[1..MAXSIZE] of integer;

   lps
      UNSORTED : in,out : (VECTOR, integer, integer);
      SORTED   : in,out : (VECTOR);
      REQUEST  : out    : (byte);
      RELEASE  : out    : (byte);
      RESOURCE : in     : (byte);

   var
      A   : VECTOR;     (* the data array                *)
      M,N : integer;    (* range of partition to be sorted *)

   procedure QUICKSORT(M,N : integer);

      const
         SERVER    = 100;      (* process ID of server process   *)
         SERVERREQ = 1;        (* external LPN in server process *)
      var
         I,J,PIVOT,TEMP  : integer;
         B               : VECTOR;    (* to hold temporary result *)
         PROCESSID       : byte;      (* ID of allocated process  *)

      begin
         if M < N then begin
            I := M;  J := N+1;  PIVOT := A[M];
            repeat
               repeat I := I+1 until A[I] >= PIVOT;
               repeat J := J-1 until A[J] <= PIVOT;
               if I<J then begin   (* swap A[I] with A[J] *)
                  TEMP := A[I];  A[I] := A[J];  A[J] := TEMP;
               end
            until I >= J;
            TEMP := A[M];  A[M] := A[J];  A[J] := TEMP;

            send to SERVERREQ at SERVER via REQUEST : (RESOURCE);
            receive from SERVER via RESOURCE : (PROCESSID);
            send to UNSORTED at PROCESSID via UNSORTED :
                                                  (A | M | J-1);

            call QUICKSORT(J+1, N);
```

```
            receive from PROCESSID via SORTED : (B);
            for I := M to J-1 do A[I] := B[I];
            send to SORTED at &SENDER via SORTED : (A);

         end
      end;   (* procedure QUICKSORT *)

   begin   (* Quicksort main *)

      read(SIZE);
      for I := 1 to SIZE do read(A[I]);
      A[SIZE+1] := MAXINT;

      call QUICKSORT(1,SIZE);

      for I := 1 to SIZE do write(A[I]);

   end.   (* Quicksort main *)
```

```
program QUICKSORTWORKER;

   const
      MAXSIZE  = 1000;
      UNSORTED = 1;    (* logical port numbers *)
      SORTED   = 2;
      REQUEST  = 3;
      RELEASE  = 4;
      RESOURCE = 5;

   type
      VECTOR = array[1..MAXSIZE] of integer;

   lps
      UNSORTED : in,out : (VECTOR, integer, integer);
      SORTED   : in,out : (VECTOR);
      REQUEST  : out    : (byte);
      RELEASE  : out    : (byte);
      RESOURCE : in     : (byte);

   var
      A   : VECTOR;    (* the data array               *)
      M,N : integer;   (* range of partition to be sorted *)

   procedure QUICKSORT(M,N : integer);

      const
         SERVER    = 100;     (* process ID of server process   *)
         SERVERREQ = 1;       (* external LPN in server process *)
      var
         I,J,PIVOT,TEMP : integer;
         B              : VECTOR;   (* to hold temporary result *)
         PROCESSID      : byte;     (* ID of allocated process  *)

      begin
         if M < N then begin
            I := M;  J := N+1;  PIVOT := A[M];
            repeat
               repeat I := I+1 until A[I] >= PIVOT;
               repeat J := J-1 until A[J] <= PIVOT;
               if I<J then begin   (* swap A[I] with A[J] *)
                  TEMP := A[I];  A[I] := A[J];  A[J] := TEMP;
               end
            until I >= J;
            TEMP := A[M];  A[M] := A[J];  A[J] := TEMP;

            send to SERVERREQ at SERVER via REQUEST : (RESOURCE);
            receive from SERVER via RESOURCE : (PROCESSID);
            send to UNSORTED at PROCESSID via UNSORTED :
                                                  (A | M | J-1);

            call QUICKSORT(J+1, N);

            receive from PROCESSID via SORTED : (B);
```

```
        for I := M to J-1 do A[I] := B[I];
        send to SORTED at &SENDER via SORTED : (A);

      end
    end;    (* procedure QUICKSORT *)

  begin   (* Quicksort worker *)

    repeat

      wait(lport=UNSORTED);
      receive from &SENDER via UNSORTED : (A | M | N);

      call QUICKSORT(M,N);

      send to SORTED at &SENDER via SORTED : (A);

    until FALSE

  end.    (* Quicksort worker *)
```