1

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. Please read the authorization forms which accompany this thesis.

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. Veuillez prendre connaissance des formules d'autorisation qui accompagnent cette thèse.

**THIS DISSERTATION
HAS BEEN MICROFILMED
EXACTLY AS RECEIVED**

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

National Library
of Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

THÈSES CANADIENNES
SUR MICROFICHE

0-315-22198-4

NAME OF AUTHOR/NOM DE L'AUTEUR Christopher Geoffrey Griffiths

TITLE OF THESIS/TITRE DE LA THÈSE Protocol Software Portability

UNIVERSITY/UNIVERSITÉ Carleton University

DEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE Master of Computer Science

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE GRADE 1984

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE Prof. R. J. A. Buhr

Permission is hereby granted to the NATIONAL LIBRARY OF
CANADA to microfilm this thesis and to lend or sell copies
of the film.

The author reserves other publication rights, and neither the
thesis nor extensive extracts from it may be printed or other-
wise reproduced without the author's written permission.

L'autorisation est, par la présente, accordée à la BIBLIOTHÈ-
QUE NATIONALE DU CANADA de microfilmer cette thèse et
de prêter ou de vendre des exemplaires du film.

L'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 l'autorisation écrite de l'auteur.

DATED/DATE Jan 31 1984 SIGNED/SIGNÉ Chris Griffiths

PERMANENT ADDRESS/RÉSIDENCE FIXE 125 Clarendon Ave. Ottawa, Ont. Canada

K1Y0R2

2

PROTOCOL SOFTWARE PORTABILITY

by

© Christopher Geoffrey Griffiths

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of
Master of Computer Science

Department of Systems and Computer Engineering

Faculty of Engineering

Carleton University

Ottawa, Ontario

December 1983

The undersigned recommend to the Faculty of Graduate Studies and Research, the acceptance of the thesis,

"PROTOCOL SOFTWARE PORTABILITY"

submitted by Christopher Geoffrey Griffiths, B.Sc. Physics and Computer Science, submitted in partial fulfillment of the requirements for the degree of Master of Computer Science.

Thesis Supervisor

Chairman,

Department of Systems and
Computer Engineering

Carleton University

Abstract

This thesis develops principles for the structuring of portable protocol software. The issues involved in applying these principles to a practical implementation are also covered.

A comparison is made of several protocol software organizations, with the intention of defining a structure that is portable with respect to operating systems. The term "structure" is used to imply the way in which the protocol software is organized into modules, together with the manner in which the modules interact. A message passing structure based on table driven protocol modules is described and justified.

In order to investigate the practical use of the developed structuring principles, they were applied to the design of an OSI gateway. The implementation of the transport layer of this gateway is covered in detail. This applied study leads to the formation of strategies for the practical structuring of protocol software.

Acknowledgements

I would like to thank Dr. R.J.A. Buhr, my supervisor, for suggesting this research topic, for organizing and supervising the related Teletex project and for providing invaluable advice and guidance. His enthusiasm and insight were sincerely appreciated.

I would also like to thank the other graduate student members of the Teletex project team, Alexander Pepple and William Malek. They contributed to discussions which generated many original ideas.

I take this opportunity to thank Gerald Karam for offering his valuable advice and experience in technical writing.

Finally I would like to thank Brigitta Griffiths for helping with the proof reading of this thesis.

To my parents,

Table of Contents

Abstract	i
Acknowledgements	ii
Table of Contents	iii
List of Figures	v
 CHAPTER 1 INTRODUCTION	 1-1
1.1 Background	1-1
1.2 Gateway Project	1-1
1.3 Objectives	1-4
1.4 Outline	1-5
 CHAPTER 2 APPROACHES TO PROTOCOL SOFTWARE ORGANIZATION	 2-1
2.1 Introduction	2-1
2.2 Layering Principles	2-5
2.2.1 Introduction	2-5
2.2.2 Nature of Layer Interactions	2-6
2.2.7 Software Structures for Layering	2-7
2.3 Conclusions	2-20
 CHAPTER 3 ISSUES AND APPROACHES FOR PORTABILITY	 3-1
3.1 Introduction	3-1
3.2 Levels of Software Portability	3-2
3.3 Approaches for Portable Implementations	3-4
3.3.1 Introduction	3-4
3.3.2 The Highest Common Factor Approach	3-5
3.3.2.1 Introduction	3-5
3.3.2.2 Porting Between Sequential and Multitasking Environments	3-5

3.3.2.3	Porting Between Different Multitasking Environments	3-10
3.4	Modularization	3-13
3.5	Multiple Channel Layers	3-15
3.6	Conclusions	3-19

CHAPTER 4 APPLICATIONS OF THE APPROACH

4.1	Introduction	4-1
4.2	Teletex Software Development	4-4
4.2.1	General Requirements	4-4
4.2.2	UNIX Oriented System Design	4-6
4.2.3	Implementation Details	4-13
4.3	Portability - Performance Tradeoffs	4-21
4.3.1	Throughput in UNIX and RMX	4-21
4.3.2	Space Efficient FSM Coding	4-28
4.3.3	Portability versus Efficiency	4-30
4.4	Discussion	4-32

CHAPTER 5 CONCLUSIONS 5-1

References

APPENDIX A	UNIX Performance Test Program
APPENDIX B	iRMX/86 Performance Test Program
APPENDIX C	Teletex Transport Layer Timing Test Program

List of Figures

2.1	UNIX Structures	2-2
2.2	Ada Structures	2-2
2.3	Concurrent Pascal Structures	2-2
2.4a	Sequential / Procedure Calls Structure	2-9
2.4b	ISO Class 0 Transport Interface	2-9
2.5a	Ada Task Messaging Structure	2-11
2.5b	Layered Monitors Structure	2-13
2.6	Sequential / Messaging Structure	2-15
2.7a	Multitasking / Procedural Structure	2-18
2.7b	Grouped Service Primitives	2-19
3.1	Pseudo-Process Structure	3-8
3.2	Ada Protocol Service Module Structure	3-9
3.3	Multichannel Layer	3-16
4.1	Gateway Environment	4-2
4.2	Layered UNIX Processes Structure	4-7
4.3a	UNIX Teletex Process	4-9
4.3b	UNIX 4.2 Environment	4-9
4.4	Pseudo-Process Internals	4-12
4.5	Protocol Service Module in Procedural Context	4-15
4.6	Teletex Transport State Transition Diagram	4-17
4.7	Protocol PSM Modularization	4-18
4.8	Modified Data End-to-end State Internals	4-20
4.9	Test Software Structure	4-23
4.10	Graph of Execution Time versus Number of Sublayers under UNIX 4.1c	4-24
4.11	Graph of Execution Time versus Number of Sublayers under iRMX/86	4-25

Chapter 1

INTRODUCTION

1.1 Background

In the field of computer communications, the portability of software can be particularly valuable. Protocol software has the unique property that the service it provides must be supported on two or more machines before it is used. As protocol standards develop and become accepted, portable implementations will be required for widespread use in diverse environments. Descriptions of portable protocol software are beginning to emerge in the technical literature [2]. However, little work has been reported on the basic software structuring techniques for portability [11].

1.2 Gateway Project

The material presented in this study evolved during the course of an ongoing project to develop an OSI session level gateway between Carleton University's local area network and Datapac. The project involved a team, including the author and two other graduate students, supervised by R.J.A. Buhr. The ultimate goal for the development of this

research tool was to enable communication with several interested organizations including the Department of Communications, the University of Montreal and the University of Ottawa. As these sites support different programming and operating environments, software portability was important for exchanges of protocol implementations to take place. The gateway project involved the design and implementation of the session and transport layer protocols of the Trans Canada Telephone System Teletex protocol specification. This software will run under Berkeley 4.2 UNIX and will be interfaced to X.25 driver. One motivation for having this communications interface is that the other, above listed interested organizations are involved with similar Teletex-X.25 projects.

Another motivation is the relationship of Teletex to the reference model of Open Systems Interconnection (OSI) of the International Standards Organization (ISO). The Teletex S.70 transport layer specification is based on the proposed ISO Class 0 transport protocol. The Teletex S.62 session layer specification is part of a proposed ISO session layer protocol standard.

The justification for the implementation of a "high level" (transport / session layer) gateway is that the network interfaces to each of the end networks exist or are readily available. The design and implementation phases of

the development of the gateway software have both generated and been affected by considerations of portability. These phases and the portability related factors are covered in detail in the subsequent chapters.

The current state of the project is as follows:

- Functional requirements and system architecture specification complete [14].
- User and inter-layer interface specifications complete [14],[15].
- Transport layer implemented and tested [21].
- Session layer partly coded.

Specifically, the work contributed to the gateway project by the author includes:

- Participation in the team-design of the gateway functional requirements, system architecture and layer interface specifications.
- Compilation and composition of the Teletex Package User's Manual [15].

- Coding of the ISO Class 0 Transport layer.
- Co-authorship with R.J.A. Buhr of a report on portable protocol software organization [11].

The following items remain outstanding for completion of the gateway project:

- Coding and testing of the session layer.
- Implementation and testing of an interface to X.25.
- Implementation and testing of a demonstration system.

1.3 Objectives

This thesis has two objectives. The major objective is to develop strategies for software structuring in the design of portable OSI software. The second objective is to examine the issues involved in the realization of a protocol implementation using the developed strategies.

1.4 Outline

Chapter 2 introduces the topic of protocol software organization. A graphical technique used to illustrate software structures is defined. Various organizations are presented, and their implementation is discussed.

Principles for the organization of portable layered software are developed in Chapter 3. First, the meaning and context of the required portability is examined with respect to other types of portability. Then a portable software structure is presented and its internal modularization is described. The application of the principles used for single channel protocol software design, to multichannel software organization is outlined.

Chapter 4 examines the practical issues in the use of the organizing principles and relates the experience drawn from the gateway project. Design decisions and implementation details as well as performance issues are covered.

Chapter 5 summarizes the conclusions of this thesis with respect to the objectives. The implications towards related areas are discussed. Suggestions for further work are made.

CHAPTER 2

APPROACHES TO PROTOCOL SOFTWARE ORGANIZATION

2.1 Introduction

This chapter provides an overview of various protocol software organizations and design issues in order to provide a framework for the subsequent discussion on portability.

Section 2.2 gives an overview of the layering principles involved, and outlines the various logical system structures appropriate for single channel protocol layers. The implementation of these structures is outlined with respect to various operating systems.

Throughout this thesis, structure graphs defining software system structures will be used. To complete this introduction, a definition of the structural notation follows.

Graphical Notation for System Description

The technique of graphically illustrating the structure of software has been proven valuable for both conveying information and solving design problems. Design

Figure (2.1)

UNIX STRUCTURES

Figure (2.2)

ADA STRUCTURES

Figure (2.3)

CONCURRENT PASCAL STRUCTURES

level graphical representations of concurrent systems can provide a quick, intuitive understanding of their operation. Such a graphical notation has been presented by Buhr [13], using ADA for system design specification. Many of the examples in this thesis will be illustrated using the notation developed by Buhr. Although some of these examples will be drawn from other environments, such as UNIX or Concurrent Pascal, a similar notation will be used for these. The structure graphs in Figures (2.1) through (2.3) depict fairly arbitrary structures from these environments, but illustrate most of the features used throughout the subsequent chapters.

In all situations, the autonomous entities, UNIX processes, ADA tasks etc. are shown as parallelograms. Figure (2.1) depicts two UNIX processes A and B which both access a UNIX pipe. The connecting arrows represent access to the pipe only. The small arrows with circles at their tails represent data flow during the pipe access. Process A is reading from the pipe, and may be blocked if the pipe is empty. This possible suspension is denoted by the dot beside the pipe access arrow. Process A contains a private procedure represented by rectangle C. Process B contains a private data structure D shown as a circle.

Figure (2.2) illustrates the features of ADA which are referenced in the structure graphs of this thesis [13],[20].

The large rectangle C represents an ADA package, with an interface procedure D. This is an active package, containing an ADA task B. The external ADA task A calls the package interface, exchanging both input and output data which are represented by the same arrows as in Figure (2.1). The interface procedure then may call the entry E of task B. The line enclosing the entries E and F of task B indicates that they are part of an ADA selective accept statement. The dot beside entry E means that it may be guarded, causing task A to be blocked until the guard is removed.

Figure (2.3) contains structures found in the Concurrent Pascal language. The notation used here is taken from the book by Bowen and Buhr called "The logical Design of Multiple Microprocessor Systems" [19]. Process A makes a call to the entry procedure B of the monitor C. The double walls of the monitor rectangle distinguish it from a procedure or package and represent the property of exclusive execution of the calling processes. The dot beside the procedure access arrow implies that the calling process may be suspended on an internal queue of the monitor.

Certain features will be common to all structure graphs in this thesis, regardless of the defined operating environment. Autonomous entities will be shown as parallelograms. Passive structures containing executable code (such as procedures or packages) will be illustrated as

rectangles. Data structures will be shown as circles or ovals, and the arrows connecting these elements will always denote the direction of access.

2.2 Layering Principles

2.2.1 Introduction

Protocol layering is a basic principle in computer communications. SNA, ARPANET, and X.25 are common examples of layered network protocols. This thesis is centered around the seven layer Reference Model of Open Systems Interconnection (OSI) developed by the International Standards Organization (ISO).

The layering concept can be divided into two basic areas:

- Data Stream Layering, and
- Logical System Structure Layering.

Data stream layering refers to the nesting of higher level protocol data units (PDU's) as the data passed in lower level PDU's. This approach is part of the OSI model, and refers strictly to the end-to-end protocol data stream. There are no implications here towards the software

organization or local (inter-layer) data flow.

Logical system structure layering is the basis of most protocol software designs. The software is modularized in a layered fashion with respect to functionality and level of abstraction. The OSI reference model provides a minimal division of functions upon which a logical system structure can be based. From this point of view, a layered architecture is a natural conclusion, although not necessary for OSI compatibility. Layered protocol software design is one of the main concerns of this study.

2.2.2 Nature of Layer Interactions

Without considering the actual definition of the software structure, for a layered system, the nature of the layer interactions can be investigated. Two approaches are possible:

1. - Interactions are processed as they arrive.
2. - Interactions are blocked until a suitable state is reached, and then processed.

Each of these approaches affects the outcome of the design of a system structure. If Approach 1 is adopted, a message passing system would be appropriate. A layer would

process any arriving message and respond by sending messages to the neighbouring layers. If Approach 2 is taken, interactions would take the form of procedure-like calls between layers. In this way, an interaction call returns with a final result. Layered system structures using these approaches are developed and compared in the next section.

These two approaches for interaction management also affect the functionality of the service interfaces provided by the layers. For example, if interactions are always processed as they arrive, a credit mechanism for inter-layer data flow might be necessary to avoid the flooding of a layer with data interaction requests.

2.2.3 Software Structures for Layering

Depending on the intended environment of the software implementation, various possible software structures for general layered systems are feasible. Research into layered software structures has been carried out at Carleton University by R.J.A. Buhr, [20]. In the light of the available current software tools, there are two general views which categorize layered structures and their inner workings:

- View 1. Sequential versus Multitasking
- View 2. Procedure-like Calls versus
Interaction Messages

These views can be combined to produce the four basic logical system structures illustrated in Figures (2.1) to (2.4).

Structure 1. Sequential/Procedure Calls Structure

The Sequential/Procedure Calls structure is depicted in Figure (2.4a). Each protocol layer is composed of a collection of passive procedures. The diagram depicts the layers as ADA packages with internal static data for protocol states, but this is not necessary. This kind of structure could be implemented in a Pascal-like language with the current state data being returned to the calling layer and to be used as input in subsequent calls. The simplicity of this structure is appealing, but several drawbacks can be found if the operation of such a layered system is examined.

The service interface of an ISO Class 0 Transport layer implementation following this structure is illustrated in Figure (2.4b). Each interface procedure corresponds to a local protocol service primitive.

Figure (2.4a) Sequential / Procedure Calls Structure

Figure (2.4b) ISO Class 0 Transport Interface

Because of the hierarchical flow of control, full duplex communication is difficult. The specification of this protocol states that in the data state, data PDU's can be both sent and received. To service a receive call, the transport layer must poll the network until a packet arrives. If the transport layer does not return program control to the session layer until data is received, the entire layered system including the application is held up. An unelegant solution to this problem is to inquire about the arrival of transport PDUs on every transport service request. It is then the responsibility of the session layer to make a receive call. Unfortunately this mixes the functionality of the individual service requests. For example, if the calling session layer is continually sending data, arriving PDU's will be lost unless their arrival is flagged in the Send parameters.

Structure 2. Multitasking/Interaction Messages Structure

One Multitasking/Interaction Messages structure is shown in Figure (2.5a). Here, each layer is composed of a single ADA task and a mailbox active package for the interaction messages. In concurrent Pascal this could be translated to processes with suitable mailbox monitors. In iRMX 86, tasks with mailboxes would be used. In UNIX, processes connected by pipes would be appropriate.

Figure (2.5 a) ADA Task Messaging Structure

Obviously, this structure is portable to any multitasking environment, and as is shown in the next basic structure category, even to sequential (non-multitasking) environments. A typical layer process in such a system operates as follows. An interaction from one of the neighbouring layers is received from the mailbox after a possible wait. The interaction is evaluated and resulting local interactions are sent to the neighbouring mailboxes. Each layer process can maintain static state variables and provide independently initiated events such as timeouts. Clearly, full-duplex data flow is easily accommodated.

The duality of monitors and processes has been suggested by Cashin [7], and can be utilized to generate the corresponding structure of Figure (2.5b). Here, each protocol layer process has been replaced by a monitor, and the mailbox monitors by transport processes. The transport processes perform the simple repetitive task of waiting for an interaction to be provided by one layer and then passing it on to the next. As long as a suitable local credit mechanism is employed, these transport tasks should only be suspended when no interactions are ready for pick-up but not during the delivery phase. In analogy to their process counterparts, the monitors operate on a per local interaction basis. For example, when a transport task calls the DEPOSIT entry procedure, the interaction is evaluated and if an upwards-flowing service interaction is the result,

Figure (2.5b) Layered Monitors Structure

the RCV queue is signalled. If a downwards-flowing interaction results, the GET queue is signalled.

As pointed out by Cashin [7], such dual systems roughly conserve performance properties such as queue lengths, waiting times and number of context switches, and are logically equivalent. However, it may be preferable from a software design point of view to choose the mailbox monitor structure. The mailbox monitor structure is likely the most portable one with respect to multitasking operating systems. Also, a standard software engineering practice in the design of such systems is to assign the distinct functional tasks (in this case the layered protocol services) to be performed by individual processes, with monitors used to coordinate their sharing of data and resources. Timeouts could be handled in this case by providing a "tick" entry procedure in each monitor, with individual clock processes to call each one periodically. Again, bidirectional flow can be readily supported.

Structure 3. Sequential/Interaction Messages Structure

The Sequential/Interaction Messages structure is shown in Figure (2.6). Each layer consists of a passive protocol package or procedure, and mailbox package. The main program schedules the operation of the layers as follows. If there is an interaction to be processed in one of the mailboxes,

Figure (2.6) Sequential / Messaging Structure

the corresponding layer is activated. The layer procedure retrieves the interaction message, processes it, and sends resulting interactions to the neighbouring mailboxes. Control is then returned to the main program along with parameters containing the updated status of the mailboxes. If the operating environment is not a multitasking one, all protocol software layers would be included as layer procedures. The network interface is intentionally ignored in this description, and will be examined more closely in the subsequent chapters.

Efficiency is also affected in this design. The ability to pass pointers between concurrent processes is not found in many multitasking operating systems, thus causing delays in multilayered data flow. This aspect will be discussed in more detail in Chapter 4. The layer protocol procedures function much like the tasks of Figure (2.5a), and this property has been utilized in portability applications (Chapter 4). This structure could be implemented in any high level language. Also as is shown in subsequent chapters, the structure, logic, and even source code of the layers is portable to multitasking designs and operating systems. In comparison with the Sequential/Procedure Calls structure, the mailbox framework of this design may introduce some overhead to the overall processing, but in addition to the portability aspects, other problems are overcome. Timeouts can be handled by

setting up the main program to provide periodic timing interaction messages to the layers. Also, full duplex data flow is now a more natural operation for the system.

Structure 4. Multitasking/Procedure Calls Structure

The final logical system structure to consider is the Multitasking/Procedure Calls combination. The Procedure Calls aspect implies that the service primitives of each layer should be provided as monitor entry procedures (or ADA task entries), as opposed to the method of using a service identifier in an Interaction Message.

As is illustrated by Figures (2.7a) and (2.7b) this requirement leads to overcomplicated layer structures which are logically similar to the simpler messaging alternatives. Figure (2.7a) shows a Session-Transport interface in this structuring context. Each layer consists of an ADA task providing entries for each service primitive at both local interfaces. A set of simple relay tasks (not necessarily equal in number to the number of entries) is used to bridge each interface.

In comparison with the Multitasking/Interaction Messages structure, this design has no advantages except that messages do not have to be decoded and encoded. The

Figure (2.7a) Multitasking / Procedural Structure

Figure (2.7b) Grouped Service Primitives

large number of tasks and entries makes the structure overcomplicated and wasteful of resources for our application. The main logical difference from the messaging structures is that interactions can be blocked by the layer, for example if the send buffer is full. This flexibility is inherent to the procedural approach in general. Both models can be easily extended to handle multiplexing of several layer entities.

If the service primitives are grouped into requests, confirms, indications, and responses, a structure such as Figure (2.7b) can be obtained. Further, a structure such as the one in Figure (2.5b) can be obtained by grouping primitives at which processes wait for data, and combining those at which data are deposited. Now, the identification of interaction types is based solely on message information, as opposed to structural interfaces.

2.3 Conclusions

A hierarchical system structure fits naturally into the OSI reference model framework. Four basic layered system structures have been discussed, based on two categorizations of software in general. The portability of each system structure has been introduced, and this area is explored in the following chapters. However, already

examples of the similarity in structure and functionality of message passing software entities have been given. This concept will become a basis for the design of portable logic system structures.

CHAPTER 3

ISSUES AND APPROACHES FOR PORTABILITY

3.1 Introduction

This chapter investigates protocol portability issues from various viewpoints. The object is to define generic portable software structures. Much of this material is derived from the practical experience in the design of the gateway software. Consequently the emphasis is on single channel type layers, although multiple channel architectures are not excluded from the discussion.

Section 3.2 reviews the various degrees to which software in general can be made portable.

Section 3.3 approaches protocol software design from various directions to obtain guidelines for layer software organization.

Section 3.4 investigates multiple channel designs and relates the approaches from Section 3.3 to these ideas.

Section 3.5 draws conclusions about the issues raised in this chapter and summarizes a single channel protocol layer software structure.

3.2 Levels of Software Portability

As a framework for this discussion a definition or scale of software portability would be ideal, but such a definition cannot be precise. However, the various ways that software can be transported between two environments can be related with respect to portability. This section is particularly concerned with some of the basic levels of software portability found in practical applications.

A completely portable piece of software is one that runs identically in both environments with no changes to itself or to the new operating system. This type of portability is only found in the cases of extremely simple, high level software or essentially identical environments, as would be the case with hardware "clone" machines running the same operating system.

A slightly lower level of portability is found in cases where only the machine architecture dependent portions of the software must be rewritten. For example, addresses of device ports may differ making new device drivers necessary. Changes such as these usually involve no modifications to any high level source code, and are relatively straightforward.

A common problem to be overcome will often be a

difference in high level language compilers. The portability characteristics of the compiler used in the implementation are important. Basically, high quality, popular compilers should be used. Minor alterations will often be necessary, but complete language translation may sometimes be required. Even when automated language translation is available, human intervention is always necessary, and in some cases the translation is impossible. When it is possible that these types of problems may be encountered, only the most common language features should be used in the implementation.

The differences between operating systems may pose the most difficult problems in the porting of software. For portability, the operating system-dependent portions of the software should be separated so that they can be replaced in a modular fashion. This type of modularization is reflected in the early design of the logical software structure.

The porting of logical structure implies the lowest degree of software portability since this is the earliest stage in the implementation. Thus, if portability is required, careful consideration should be given to making the logical structure of the software itself portable. The approach taken in this thesis will be to investigate structuring techniques for portability. In specifying portability as a basic requirement, a secondary requirement

is implied in that full advantage be taken of the facilities provided by the different operating systems. The issues involved in the design of such structures are discussed in the following sections.

3.3 Approaches for Portable Implementations

3.3.1 Introduction

The intent in this section is to develop software structuring techniques for the design of portable protocol implementations. Two standard approaches for software portability are taken in the development of these methods. First, the "Highest Common Factor" approach is investigated. This approach leads to the concept of the "Protocol Service Module", and to the favouring of message passing models. Next, modularization in several contexts is examined, with several general concepts emerging, including again the Protocol Service Module. Multiple channel protocol designs are then looked at in the light of the approaches developed thus far.

3.3.2 The Highest Common Factor Approach

3.3.2.1 Introduction

When portability between two or more operating environments is required, the use of only the common services, or features of these is an obvious measure. In many cases this proves easier than constructing translations of the special properties of some unique operating system. Therefore, if portability is a high priority, not even the early structural designs of software should be made without some consideration of the possible application environments.

3.3.2.2 Porting Between Sequential and Multitasking Environments

One of the basic requirements in transporting software from a sequential to a multitasking environment must be that the new system take full advantage of the multitasking facilities. In other words, translating the original program into a single autonomous process is generally not sufficient. Even so, the only common features found between such environments are the usual high level language constructs. This implies that the protocol services of any one layer should be structured as a procedure, or package of procedures. In the foregoing (Chapter 2), two alternative

sequential structures for layered designs are outlined.

The Sequential/Procedure Calls structure of Figure (2.4a) has several drawbacks with respect to this type of portability. Because of the hierarchical flow of control, there would be no point in converting each layer directly into an autonomous entity. Also, any one layer could not be easily used in this way, to be configured as part of a system of layered message passing processes. The reason for this is that in order to process a service request, a call must be made to some lower level service, with no return until the service is completed. Thus the layer is hung-up, polling its mailbox for a specific response from the lower layer. Unexpected interactions are difficult to process.

Fortunately, the Sequential/Messaging Structure of Figure (2.6) possesses the required portability to multitasking structures. As noted in Chapter 2, the protocol layers of this structure function much like the message passing tasks of Figure (2.5a). For this reason, such layers will be referred to as pseudo-processes. Any one of these layers could be converted into a message passing type process by simply changing the procedural accesses of the mailbox packages into interprocess communication calls, and enclosing the code in a forever looping process structure. In order to make this

transformation as simple as possible, the actual protocol service code should be cleanly separated from the code which accesses the local operating environment. Therefore, the pseudo-processes should contain a Protocol Service Module (PSM) as is shown in Figure (3.1). The pseudo-process is a passive package with an interface procedure `N_PI` (for N-layer pseudo-process interface), and the protocol service module procedure. The interface procedure provides the compatibility between the protocol service module which is now operating system independent, and the local environment. This is the part of the pseudo-process which would be modified to contain the interprocess communication calls and process-like looping structure. As an example, Figure (3.2) shows the structure of a layered system of ADA tasks with the same protocol service modules as the typical pseudo-processes. It should be noted that in addition to the layer interface procedures, the mailbox skeleton must be recreated when moving between these environments.

The protocol service module operates on a per local interaction message basis. It accepts as input one interaction, and gives as a result either one or two interactions, and their destinations (layer N+1 or layer N-1). The interface procedure is responsible for delivering them. The protocol service module is developed further in the next sections and in the subsequent chapters.

Figure (3.1) Pseudo-Process Structure

Figure (3.2) ADA PSM STRUCTURE

3.3.2.3 Porting Between Different Multitasking Environments

In considering the transport of software between different multitasking environments, the portability is dependent on the common supported features of these. For example, ADA is regarded as being useful as a software design specification tool. This is because the multitasking operating system functions are incorporated into the basic language constructs (tasks, entries etc.). However, for the specifications to be portable to other operating systems, the language constructs must also be meaningful in those contexts.

The basic task and rendezvous concept of ADA is certainly portable to most operating environments. The rendezvous mechanism can be imitated in mailbox messaging systems such as iRMX 86 with the use of mailbox pairs for each task call/entry combination. However, other important features of ADA are much less portable. For example, the selective accept mechanism is more complicated to arrange for a UNIX process. In standard UNIX, the implementation of a system using selective accepted rendezvous constructed with pipes is not trivial. The reason for the difficulty is that a UNIX process cannot inquire about the identity or existence of information in a pipe. It must read the pipe, extracting the data, and is suspended if the pipe is empty

until another process writes some data to it. This implies that the use of pairs of pipes between layers would not be sufficient. If a layer process is waiting at a pipe at one interface, it is ignoring the other interface. Thus in the particular case of standard UNIX, the single mailbox (pipe) per layer structure is the best alternative. This is an example of the effect of the application environment on the early structural design of a system.

This problem is simplified in iRMX/86. A receive call made to an empty mailbox can impose an optional wait for the arrival of data, or return immediately or after a specified time with a condition code parameter. Therefore a task could poll various mailboxes if necessary.

In the transformation of an Ada task to a "monitor dual" as referenced in section 2.2.3, the selective accept mechanism is portable. Each selectively accepted entry can be mapped into monitor entry procedure with an internal condition variable queue.

In the implementation of an Ada design in another environment, rendezvous emulation is not required, provided a messaging system was used. If Ada tasks are connected by mailbox tasks or active packages as in Figure (3.2), the design will be portable.

The mailbox structure is particularly portable between the common existing multitasking operating systems. The main characteristics that may vary are the ability to pass pointers to shared data between processes, and the option of not waiting if the mailbox is empty. Those systems which do not explicitly provide a mailbox-like structure usually support sufficient interprocesses communications to easily implement one. The portability of the mailbox structure leads naturally to the message passing model for layered systems. With the message passing model, the transformation between different multitasking systems is analogous to moving from the sequential environment to multitasking. The protocol logic should be separated from the local operating system-dependent logic in the same way. Again the protocol service module is necessary, with the local interface consisting of the skeleton process, its interface to the mailbox system, and the mailbox system itself. Section 3.3 on modularization develops the protocol service module concept in more detail.

3.3 Modularization

Modularization is a standard practice in software design, and for general software portability. With respect to portability, the basic idea is to cleanly separate the portable portions of the code from the unportable parts. In this way those portions which may require changes for different environments can be replaced in a modular fashion. For example, the modularization of the pseudo-process structure (Figure (3.1)) separates the unportable interlayer communications from the portable protocol services (PSM). Another function of this separation is to make possible the configuration of various systems of these layers. The protocol service modules do not require any information about the identity of the neighbouring layers because the interface procedure or calling process performs the actual data transfer. The calling process (or interface procedure if not in a multitasking environment) maps the up or down destination information from the protocol service module into mailbox identifiers. This mapping could be established in initialization time parameters of the system. The configuration of different systems out of node layers is desirable from the points of view of constructing test systems, or building nodes out of "off-the-shelf" software. Obviously for this type of configurability, software portability is crucial.

As was pointed out earlier (Section 3.2), the degree of portability depends largely on the characteristics of the programming language used. This implies that a popular, high level language would be suitable. This is a highest common factor type of argument. Unfortunately, such languages often do not provide the necessary facilities to implement a communications interface. It may be necessary to create the required functions, which would be accessed through calls to separately compiled, possibly assembler coded modules. The calls, would be made from the layer interface procedure and main program, or the PSM-calling process. Along with the functional add-ons, these calls would add to the system dependent, unportable portion of the code.

To be able to configure systems with layers from different sources, the local (inter-layer) protocol should be coded in a flexible or replaceable manner. These kinds of modifications must be performed without affecting the end-to-end protocol management. It is thus necessary to have a clean separation of the local protocol logic from the end-to-end protocol control. A local protocol module could then be completely replaced if necessary. If the protocol finite state machines (FSM) are table driven, changes to individual transitions and actions can be straightforward. However, in practice it may be difficult to completely separate the end-to-end and local FSM's. An example of this

difficulty is explored in the practical application in Chapter 4. In addition to local/end-to-end modularization, the layer's testbed logic should be further modularized. This should pose no serious difficulty since testing interactions are usually independent of the protocol FSM.

The above modularization practices are discussed with respect to multiple channel architectures in the next section.

3.4 Multiple Channel Layers

A discussion of portable multiple channel software is limited to multitasking environments, since only single channel protocols are of real use in nonmultitasking ones. A multitasking software example based on a general system proposal by Piatkowski [8], will be discussed in this section, with emphasis on the portability of message passing based designs.

Figure (3.3) illustrates a high level ADA description of a typical multichannel layer in a communications node. Each channel is supported by an autonomous N_PORT task. These port tasks are allocated, and perhaps dynamically instantiated by the N_MGR task. Note that every task in the system has a single mailbox active package which it uses to

Figure (3.3) MULTICHANNEL LAYER

receive all interactions from other tasks. The N_MGR receives the initial request for a channel from the upper layer, and if the request can be satisfied, all further interlayer communications will take place with an N_PORT task. The N_PORT tasks will be responsible for all the services that a layer task in a single channel architecture would support. These functions would include connection establishment and disconnection, end-to-end data flow management, local (interlayer) services and flow control. The N_ROUTER task performs several interrelated functions. It provides multiplexing and demultiplexing of the lower layer channels, channel routing functions, and all interlayer data transmission.

The portability of this general layer design is based on the message oriented interprocess communications. Language and operating system features are not relied upon for any functionality. As stated for the single channel architectures, a mailbox-like feature is portable to any multitasking environment. If the information communicated between tasks is based only upon specified formats of exchanged data, the portability of the design will be ensured.

All the tasks in a message passing system can be designed in the same way that the layered processes of the single channel node design were. Each such task contains an

interface to the local communications, which would be the unportable portion of the code. The tasks would loop around a central procedure, like the PSM discussed earlier, which would process and return specially formatted messages. The message based logic, if not the actual code of this procedure would be the portable portion.

In particular, the N_PORTS would each contain a protocol service module similar to that of a single channel process. The protocol service module would then be modularized with respect to the same issues of interlayer communication mechanisms, layer configurability, and local/testing/end-to-end protocol separation.

This multichannel example is an extension of a proposal by Piatkowski [8] for an OSI systems approach. The general structure of the two approaches is the same. The extensions made here are the explicit definitions of the system component types and their methods of intercommunication. However, some small differences between the underlying models exist. The manager module in Piatkowski's proposal supports end-to-end protocols between peer managers including connection establishment and error recovery functions, in addition to management and allocation of local resources. Distribution of the end-to-end protocol management between the manager and ports is not necessary, and it can be naturally supported by the port alone. Also,

if the manager is establishing connections for many processes, "bottlenecking" of their service requests is possible.

3.5 Conclusions

The portability of the logical structure of software is a basic requirement for maximizing the portability of the software itself. The use of commonly supported features is an obvious measure, and leads to a message based system of layered processes for single channel protocols. Portability between various multitasking environments as well as sequential ones is then possible.

The following list summarizes the guidelines for the design of portable protocol software structures, discussed in this chapter:

- Use of message passing layers.
- Layered pseudo-processes and mailboxes in a sequential programming environment.
- Layered processes and mailboxes in a multitasking environment.

- Pseudo-processes
  - Separation of protocol logic from O/S dependent logic.
  - Separation of protocol logic from layer connecting software.
- Protocol Service Module.
  - Portable protocol server.
  - Processes interaction messages.
  - Separation of local/end-to-end protocol logic.

These ideas can be extended to multiple channel designs, and multitasking software in general. Specifications of intertask communications based on messages are highly portable. The practical application of these ideas is developed in Chapter 4.

CHAPTER 4

APPLICATIONS OF THE APPROACH

4.1 Introduction

This chapter describes a UNIX application of the principles for portability and the practical issues involved. The results of some throughput tests under UNIX and iRMX/86 are presented as well as details relating to the implementation of a protocol FSM under the constraint of portability.

The application described in this chapter is part of the implementation of a Session level gateway between Carleton University's research local area network and Datapac. The gateway environment is illustrated in Figure (4.1).

The Session level is an implementation of the Trans Canada Telephone System (T.C.T.S.) Teletex extended S.62 specification [1]. The Transport layer implements the T.C.T.S. extended S.70 specification which is based on the proposed ISO Class 0 Transport protocol. The gateway software is being written in C, for use under Berkeley 4.2 UNIX, but will run under standard UNIX Version 7. Berkeley 4.2 UNIX is a distributed operating system. It provides a

Figure (4.1) Gateway Environment

network like interface to the Internet local area network. Remote applications will have access to the Teletex software. The Teletex software will provide the Session interface to the X.25 network board in the SUN. Therefore, because of the distributed nature of the user applications, this interface provides a gateway service.

One of our goals is to enable communication between Carleton University's local area network and other computer sites. These sites support different operating systems and languages. Thus, software portability is of high priority. This chapter describes the development of the Transport layer with respect to the issues of portability discussed so far. Portability, configurability, and extensibility were imposed as major requirements of the software. The relationship between these factors is discussed in addition to the practicalities involved.

Section 4.2 describes the approaches used in designing the software. A comparison is made of the alternative organizations and concepts leading to the final structure. Implementation details which relate to portability are included.

Section 4.3 discusses the tradeoffs between portability and the performance related issues encountered in the implementation.

Section 4.4 summarizes the conclusions and experience drawn from this practical development.

4.2 Teletex Software Development

4.2.1 General Requirements

At a basic level, portability may involve no more than choosing a widely used language and operating system. The choice of the UNIX/C combination as our end environment was made because of its widespread use. This gave us a large community for which a direct form of portability could be provided. Toward this end, only standard UNIX features were used in the refined design. It is important to note in the following design description, that specific features of UNIX lead to some of the basic structuring concepts for portability.

For reasons outlined in Chapter 3, the following principles were established as requirements of the software:

- A clean separation of the protocol logic from the operating system dependent logic.
- For configurability, separation of the protocol logic from a user configurable,

layer connecting envelope. The connecting envelope provides a mapping between the input and output ports of each layer.

- Separation of the end-to-end protocol management from the local protocol handling.

As will often be the case in protocol software implementation, extensibility is one of our higher priorities. The Teletex S.62 Session level protocol will closely resemble a more general ISO Session level standard soon to be approved. Some of the S.62 service commands contain Presentation level related arguments. Therefore in addition to extensions, there may be command parameter changes in the new standard. The S.62 protocol specification is in the form of detailed finite state machines. To provide the greatest flexibility, our protocol logic is in the form of table driven FSM's. Not only is a table easily modified, but can also be readily checked for completeness and correctness. In addition, any logic which is currently only supported by the specified default requirements can be replaced in a modular fashion.

In relation to the above issues, the efficiency of the software was of lower importance, but was not overlooked. UNIX is not well suited to time critical multitasking projects. A basic requirement was that the software be

reasonably efficient under UNIX, and also be capable of improved efficiency in more suitable environments.

4.2.2 UNIX Oriented System Design

A UNIX version of the Multitasking / Interaction Messages structure is shown in Figure (4.2). Each protocol layer is an autonomous UNIX process with an input pipe. Interactions are written to a layer's pipe by the next higher layer, and up from the next lower layer. The pipe, which has a capacity of 4K bytes, acts as a queue of service data units. The user interface process spawns the entire structure using a provided initialization interface function. Therefore if a multichannel network service is available, many parallel users may exist.

The queue-like nature of UNIX pipes implies that process interactions must be processed as they arrive. Therefore, as indicated in Chapter 2, a message passing structure is appropriate for this environment. Since the protocols involved do not require multiple channels, but provide autonomous timeouts (S.62) and infer the need for full-duplex data flow, this structure is a good candidate for use in many multitasking systems. The requirement of full-duplex data flow is inferred by the ability to continue transmitting pages while still being able to receive

Figure (4.2) Layered UNIX Processes Structure

acknowledgements of previous pages. However, the following considerations show that this may not be good structure for UNIX.

In standard UNIX, the pipe is the only medium for communication between processes. Pipes are used in a manner similar to file I/O. The sender writes its message as a string of characters to the pipe where the receiver reads it. A receiving process is suspended until the pipe is nonempty. Since UNIX processes have completely private data areas, even large blocks of data must be passed by value through pipes. Thus, the overhead of exchanging data blocks between layered UNIX processes is high. This effect is increased if credit interactions are required for regulated data flow. Because UNIX is a multiuser operating system, our software which is at the user level is further slowed. Popular techniques for real-time processing under UNIX involve modifications to the kernel [16],[17]. Such modifications add to the machine dependent portion of the software and decrease its portability.

A different structure, but one that still would be portable to real-time oriented multitasking operating systems was required. The Sequential / Interaction Messages structure (Figure (4.3a)) satisfies this requirement. The protocol layers have been consolidated into a single Teletex process. Better response time in UNIX due to reduced

Figure (4.3a) UNIX Teletex Process

Figure (4.3b) UNIX 4.2 Teletex Environment

process swapping and pipe I/O is the intended result. The Teletex process has a single input pipe into which interactions from the application and network levels are deposited. The application and network software can remain unchanged from the previous organization. As is demonstrated in Section 4.3.1, the exchange of pointers to blocks of data between pseudo-processes and reduction in number of calls to the O/S kernel can improve the efficiency of layered systems.

The incorporation of the Teletex process into the distributed UNIX 4.2 environment is illustrated in Figure (4.3b). Berkeley UNIX 4.2 provides a procedural interface to the local area network. Processes can access a full duplex communications channel called a socket. Stream sockets operate much like pipes, but provide a full duplex data medium. A server process provides a globally known socket through which interactions can be exchanged. The server process spawns the Teletex process with a UNIX fork call, and communicates with it via a pipe. The server process could spawn many Teletex processes and manage a multi-channel service. The Teletex main program makes calls to the network device driver to deposit network service data units. A separate relay process waits for interactions coming from the network driver, and passes them to the Teletex pipe. A separate process is required so that the Teletex process is not suspended by the network driver when

there may be interactions arriving at the pipe.

The modularization of the pseudo-process layers follows the prescription in Chapter 3. The relationship of the interface function `n_PI`, to the internals of the pseudo-process is shown in Figure (4.4). As an example of how the interface function `n_PI`, can be used to adapt the `n_PROTOCOL_SERVICE` module to different environments, one could consider the reimplementing of the pseudo-process as a process in UNIX. In this case, the `n_PI` function becomes a main controlling program interfaced to a UNIX pipe. Also, the mapping between a layer and its neighbours is isolated in the `n_PI` interface.

The interface to the protocol service package is the `n_protocol_service_function`. As input parameters it accepts a pointer to the incoming interaction and a pointer to an empty interaction buffer in case there are two resultant interactions. The incoming interaction is overwritten by an outgoing one. In this operating environment, the interactions are stored in buffers managed by the main controlling Teletex program. In the situation shown in Figure (4.2), the main programs of the individual layer processes would contain private buffers and provide the same interface to the protocol servers. The output parameters of the `n_protocol_service` functions include the interaction pointers and their new destinations (i.e. up, down, both,

• n_PSEUDO-PROCESS

Figure (4.4) Pseudo-Process Internals

or neither). Thus, no information about the identity of the neighbouring layers is required.

The next section elaborates on the practical side of use of the foregoing ideas.

4.2.3 Implementation Details

The Sequential / Procedure Calls layered structure was described in Chapter 2. At the outset of the implementation phase of the Teletex software, the effects of the low portability of this structure were experienced.

An existing implementation of the ISO Class 0 Transport layer was available for direct use, if possible. The decision to use the layered pseudo-processes structure had been made. Thus it would have been necessary to incorporate the existing Transport layer code into a pseudo-process. Since the existing code was a Sequential / Procedure Calls type of layer it possessed the following inescapable properties.

A request from the Session layer results in a downward call from the Transport layer to a Network interface procedure. The return of the call to the Transport layer must not be made until the request is fulfilled. Therefore, the called Transport procedure would have to somehow suspend

Figure (4.11) GRAPH OF EXECUTION TIME VS.
NUMBER OF SUBLAYERS
UNDER IRMX / 86

Figure (4.5) Protocol Service Module
in Procedural Context

protocol management from the end-to-end protocol management was discussed previously as being a basic design requirement. Separate FSM's for the end-to-end, local, and testing interaction handling would be ideal, with individual action modules for each. This situation was depicted in Figure (4.4). However, the end-to-end and local FSM specifications will usually be too closely related for total separation. As an example, the T.C.T.S. specification of the calling side of the Teletex Transport layer protocol FSM is shown in Figure (4.6) [1]. The large ovals represent end-to-end protocol states. These are connected by end-to-end state transitions shown as heavy lined arrows. The local protocol states are internal substates of the end-to-end states. Some local FSM inputs trigger end-to-end state transitions, and vice versa. Thus the complete separation of the end-to-end and local protocol FSM's is not possible.

Alternately, the modularization illustrated in Figure (4.7) was used. Each end-to-end protocol state was coded as an individual local FSM data structure, with a corresponding local state. An end-to-end state variable is also required to identify which local FSM is currently active. To complete the modularization, the end-to-end and local state transition actions must be coded as separate program modules. An FSM state transition now consists of a next end-to-end state, a next local internal state, an end-to-end

Figure (4.7) Protocol FSM Modularization

action (which results in a downward local interaction), an upflowing local interaction, and a downflowing local interaction (if there is no end-to-end interaction).

When compared with a single FSM structure, the following advantages of this kind of modularization become evident. The internal local interaction management of any end-to-end state can be altered in a modular fashion. Therefore, the local protocol can be modified independently of the end-to-end protocol. This was actually done in our implementation in order to add a local credit mechanism for data flow, and to remove the segmentation feature of the protocol. The modified end-to-end DATA state internals are shown in Figure (4.8). Another advantage of this type of FSM modularization was found during the programming of the testbed logic for the layer. For testing or monitoring purposes, the current end-to-end protocol state can be easily determined at any time.

The coding of the protocol FSM's is covered in more detail in Section 4.3.2.

Figure (4.8) Modified DATA End-to-end State Internals

4.3 Portability - Performance Tradeoffs

4.3.1 Throughput in UNIX and RMX

One of the reasons that the layered pseudo-processes structure was chosen for the UNIX environment was to increase throughput. The layered processes and pipes organization involves the overhead of many system calls for interprocess communication. The number of these calls increases linearly with the number of layers in the system. To test this line of reasoning, two test architectures were timed, running under UNIX, and for comparison, under iRMX/86. The iRMX/86 system is a real-time oriented multitasking operating system, whereas UNIX is not intended to be used for time critical applications. Each of the two test architectures consisted of an application process, a network "driver" process, and the intermediate layers. The difference between the two test programs was in the intermediate layers. In one case these layers were made up of individual processes. In the other case, a single process containing pseudo-process layers was used. The programs were run with the number of intermediate layers ranging from two to six.

The C code for the UNIX test is given in Appendix A. These programs were run, each as the only user process under Berkeley UNIX 4.1, on a SUN Microsystem 68000 based workstation. Figure (4.9) illustrates the structure of this

software. A data interaction was generated in the Application process. It was passed all the way down through the layers. Then credit for the next interaction was passed up through the layers. Therefore, even though the requirement of concurrent layers is eliminated, there is always a process executing. Each interaction consisted of 130 bytes (128 bytes data, 2 bytes header). This process was repeated 1000 times. The timer process sent special timing interactions to the Application process every 1 sec. The Application process recorded the time, and after 1000 data cycles, output the results and terminated the program. The PL/M-86 code for the iRMX/86 tests is listed in Appendix B. These programs have the identical structure and functionality to the UNIX test programs. However, mailboxes are used to pass "Token" selector pointers to shared interaction data between the layers. Also, in this case the timing interactions were generated on 0.1 sec. intervals.

The UNIX results are graphed in Figure (4.10). The results of the iRMX/86 experiment are given in Figure (4.11). The UNIX and iRMX/86 results appear very similar. In both cases, the elapsed time for 1000 data cycles of the application process increased linearly when layered processes were used as the intermediate layers. When layered pseudo-processes were used, the rate of increase of elapsed time was relatively insignificant. The overhead of system calls for interprocess communication outweighs the

Figure (4.9) Test Software Structure

a) Layered Processes

b) Layered Pseudo-Processes

Figure (4J0) GRAPH OF EXECUTION TIME VS.
NUMBER OF SUBLAYERS
UNDER UNIX 4.1c

Figure (4.11) GRAPH OF EXECUTION TIME VS.
NUMBER OF SUBLAYERS
UNDER iRMX / 86

overhead of performing the coordination of the pseudo-process layers.

These tests cannot be used as a benchmark to compare the execution speeds of the two processors involved, but may be applied as a valid comparison of the language-operating system-processor environments. Benchmark tests comparing the 68000 and 8086 processors have been carried out [9],[10], and these show the general superior performance properties of the 68000. However, the iRMX/86 operating system provides faster response for several reasons. In this case there was no time sharing of the processors between users as there must be in UNIX. Even though only one user was on the system, the background activity of UNIX was still present. Another important factor was the time-slicing of processes under UNIX. The iRMX/86 operating system leaves the scheduling of tasks entirely to the responsibility of the programmer. The context switches, and checking of states occur only when the running task is suspended, as opposed to happening at regular time intervals. Perhaps the most important reason for the relative speeds is in the method of intertask communication. Data transfer with UNIX pipes is performed as a bitwise copy, similar to file I/O. With iRMX/86 mailboxes, the equivalent to a pointer to shared data is transferred. These pointers point to data segments which can normally contain up to 64K bytes, and more if required.

The resultant data rate for the layered UNIX pseudo-processes was approximately 30K bits/sec., independent of the number of layers. Another test was performed using the actual Transport layer protocol service module alone. This test consisted of 1000 data cycles as described for the layer tests, but with real interactions. The C code for this test is given in Appendix C. The result was an elapsed time of 4 sec. This time is representative of the additional per-layer processing time that would result if real protocol layers were used inside the skeleton structures.

The approximate data rate resulting in the iRMX/86 pseudo-process test was 435 segments/sec. In view of this high speed, either of the layered pseudo-processes or layered processes structures could be used in this environment for most applications.

4.3.2. Space Efficient FSM Coding

The advantages of table driven FSM logic have been stated. The most portable and adaptable implementation of such a structure would be a finite automaton type of FSM. This means that transitions are defined in all states for all defined input events, including unexpected ones. The practical advantage of doing this is to be able to detect faulty or unsynchronized peers and adjacent layers. Also the protocol can be extended readily to accomodate new or redefined input events. This was the method first used in the Transport layer implementation. The FSM was simple and the testing was successful.

Unfortunately, the explicit definition of the state transitions for all unexpected events is repetitive and inefficient in terms of code length. For larger and more complicated FSMs such as the Teletex Session layer, this method would not be appropriate. One possible solution is to add some intelligence to the interaction decoding module. Unexpected interactions could be detected and dealt with in logic separate from an FSM defined for normal operation.

The repetitive state transitions can be categorized to form the following groups of input events which cause error and failure conditions:

(A) - Input events which are dealt with in the original FSM specification with predefined action results.

(B) - Unexpected, unsynchronized, or unidentifiable events originating at the next higher layer.

(C) - Unexpected, unsynchronized, or unidentifiable events originating at the next lower layer.

The modified FSM provides information to the decoding function about which events are valid for each state. Input events of type (A) remain valid as inputs to the FSM. Those of types (B) and (C) are handled separately by the decoding function. The following simple strategy is used. If an unexpected event arrives from the next lower layer, the service provided is assumed to have failed. A disconnect request is passed down, and the upper layer is informed of the failure. If the unexpected event originates in the higher layer, the service is not terminated. An indication of the unexpected nature of the interaction is passed up. Thus, termination of the "session" is left as the responsibility of the higher layer in this case.

With these changes to the FSM, the adaptability of the

implementation is decreased. When a change or addition of a transition is necessary, modifications to more portions of the program must be made. Also, the non-table driven portion is more difficult to adapt, and less portable than the table driven logic. It was deemed necessary to code the more complicated Teletex Session layer FSM in this compact manner. For consistency the Transport layer FSM will be altered to conform with the Session implementation strategy. Because of the readable form in which the original form of the Transport FSM was written, it totalled about 1000 lines of C. In comparison, an initial version of the compact form of the Session layer FSM [1] is about 500 lines long.

4.3.3 Portability versus Efficiency

At some time during the design of a layered system the relative importance of portability and efficiency must be determined. In many cases the required efficiency will infer the use of techniques which lead to unportable implementations.

In many circumstances, an operating system will provide an efficient facility that is not portable. An example of this would be the RMX mailbox send call which allows for a return mailbox to be specified for the receiver. For portability, this feature might be ignored,

although the efficiency would then be affected. The portable use of such a feature would be to cleanly separate the protocol logic from its access.

Some of the techniques for portability have involved separating the nonportable code from the protocol logic. The separation of operating system dependent logic from the protocol logic, and the separation of local from end-to-end protocol management were two examples. These techniques introduce extra code and processing, thus affecting both the memory requirements and the efficiency of the software. Again, the decision to incorporate portable constructs will depend on the specific functional requirements of the system.

Where the software is interfaced to a device and efficiency is a high priority, a high degree of machine architecture dependency of the software will be required. In such cases, the portability of the code may often be disregarded. However, even if the software of a layer is heavily machine dependent, a separation of the protocol logic from the machine dependent portion may still be possible. Usually, device interfaces are found at the lowest protocol levels of an implementation. Generally the importance of portability with respect to efficiency will be lower at the lower layers.

4.4 Discussion

The development of a protocol software structure and implementation under the broad constraint of portability has been presented. Under UNIX, the recommended structure is layered pseudo-processes. The arguments for this design include portability of code and structure, and efficiency. As long as a high degree of concurrency is not necessary during the data phase, the pseudo-processes structure is superior to layered processes. Rules for the modularization and efficient coding of protocol FSMs have also been presented.

The portability characteristics of the implemented Transport layer have already been utilized. In order to test the layer, a special test architecture was designed and implemented. The unmodified Transport layer was used. A description of the test layout and results is given in references [14] and [21].

In order to perform the throughput measurements, the Transport layer program was downloaded from the VAX to a SUN workstation. At that time, these machines were running different versions of UNIX. These two versions contained different versions of the C language compiler. The Transport protocol service module, containing no operating system accesses was completely portable.

CHAPTER 5 CONCLUSIONS

Several principles for the organization of portable protocol software were developed, fulfilling the first objective of this thesis.

At a high level, the portability of the layered message passing structure was demonstrated. The protocol service module, a portable protocol server was shown to be applicable to both multitasking and sequential environments. Within a multitasking operating environment, layered processes and mailboxes would be used. In a sequential programming environment, layered pseudo-processes would be the corresponding structure. It was also shown that the PSM could easily be used in a procedure oriented layered system, whereas a layer from the procedural structure does not possess this portability. The separation of the (portable) protocol logic contained in the PSM, from the operating system dependent logic in the layer interface, provided portability with respect to operating systems. This concept can be applied to the design of portable multitasking software in general. Also provided by this modularization was a separation of the protocol logic from the configurable layer-connecting software.

The internal modularization of the PSM lead to further organizational principles. The modularization of table driven protocol FSMs was investigated. A separation of the end-to-end protocol logic from the local interaction and testbed management was shown to be a necessary feature of a portable implementation. Thus, the local (inter-layer) protocol could be altered without affecting the end-to-end protocol.

These issues were also explored from a practical point of view by applying them to the design of an ISO transport layer under UNIX. Thus the second objective of this thesis, which was to relate the strategies for portability to an actual protocol implementation was met. A comparison of the throughput of pseudo-processes and layered processes messaging structures was done using iRMX/86 and UNIX. The reduction in number of system calls gives the layered pseudo-processes structure higher efficiency.

Future work in the areas discussed in this thesis would include applying the principles for portability in a practical application to the implementation of a more complicated multichannel protocol layer. The session layer of the gateway (single channel) which is currently under development will also incorporate these ideas.

An extension to the gateway project might include

the porting of the gateway transport layer to iRMX/86. Then using an available RMX implementation of X.25, a complete hardware configuration of the transport service could be realized.

Further research into the use of the portability principles with protocol formal definition technique (FDT) specifications is feasible for the following reasons. The FDT local interaction (channel) specifications are parameter oriented and can be mapped into message structures [21]. Also the FDTs are FSM oriented so that some modularization as described in this study might be applied.

A project to develop a computer aided design system for multitasking software (CADA: Computer Aided Design with Ada) is underway at Carleton University in the Department of Systems and Computer Engineering [22]. The use of the system for design and analysis of protocol software is intended. Examples of portable protocol software structures could be developed, analyzed and compared with respect to performance related issues. Finally, the incorporation of FTD specifications with the use of the portability principles into the CADA system could lead to an important extension of the project.

References

- [1] T.C.T.S. Teletex Terminal Specification,
1982/10/13, Release 3.0
- [2] Ford, W. S. Portable Implementation of Network
Architecture Layers, ACM, 1983
- [3] Grant, Hutchison, Shephard, A Gateway for Linking
Local Area Networks and X.25 Networks,
ACM, 1983
- [4] Piatkowski, T. F. The ISO-ANSI Open Systems
Reference Model - A Proposal for a
Systems Approach,
Computer Networks 4(1980) 111-124
- [5] Wallis, P. Portable Programming,
Macmillan Press, 1982
- [6] Hoare, C.A.R. Monitors: An Operating System
Structuring Concept, Communications
of the ACM, Oct. 1974, Vol. 17, No. 10
- [7] Cashin, P.M. Inter-Process Communication,
Bell Northern Research, May 1980
- [8] Dijkstra, E.W. Cooperating Sequential Processes,
Programming Languages,
F. Genuys Editor, Academic Press, 1968
- [9] Intel Co. iAPX 186, 286 Benchmark Report, 1982
- [10] Carnegie-Mellon Benchmarks, EDN, Sept. 16 1981
- [11] Buhr, R.J.A., Griffiths, C.G.,
Organizing Principles for Portable OSI Software,
Department of Systems and Computer Engineering,
Carleton University, 1983
(Submitted to the IEEE 4th International
Conference on Distributed Computing in May '84)
- [12] Buhr, R.J.A., Protocol Development for Open System
Interconnection: Research Plans for Carleton's
Local Area Network,
A Talk at DOC's University Week, June 2 1983
- [13] Buhr, R.J.A., System Design with Ada,
Prentice-Hall Inc. N.J. 1983

- [14] Buhr, R.J.A., Griffiths, C.G., Pepple, A.,
OSI Gateway Project: Functional Requirements and System Architecture,
Department of Systems and Computer Engineering Report, June 1983
- [15] Buhr, R.J.A., Griffiths, C.G., Pepple, A.,
OSI Gateway Project: Teletex Package User's Manual, June 1983 (Under revision)
- [16] Ross, Douglas J., Taylor, M.M.,
UNIX Support for Guaranteed Real Time Processing, USENIX Summer '83 Conference
- [17] Teixeira, T.J., High Speed Laboratory Data Acquisition on the MC-500,
USENIX Summer '83 Conference
- [18] Michell, S.G., Hierarchical Design with Ada for Layered Protocol Systems,
M.Eng. Thesis, Department of Systems and Computer Engineering, Carleton University
- [19] Bowen, B.A., Buhr, R.J.A.,
The Logical Design of Multiple Microprocessor Systems,
Prentice-Hall Inc. N.J. 1980
- [20] Buhr, R.J.A., A Graphical Design Notation for Modular Multitasking Systems,
Department of Systems and Computer Engineering, Carleton University report
- [21] International Organization for Standardization (ISO), ISO/TC 97/SC16/WG1 Open System Interconnection, Example of a Transport Protocol Specification, Contribution to the meeting of the WG1 ad hoc group on FDT, Nov. 1982
- [22] Buhr, R.J.A., Karam, G.M., Woodside, C.M.,
Towards Computer Aided Design of Systems to Implement Communication Protocols,
Department of Systems and Computer Engineering, Carleton University, Oct. 31, '83,
(Paper for submission to ACM SIGCOMM 84),

APPENDIX A

UNIX Performance Test Program

```

/* Per-process Global Variables */

#define DATA 0
#define CREDIT 1
#define TIME 2

int i, n, ID, EVENT_PIPE[2], UP_PIPE[2], DOWN_PIPE[2], FORK_RESULT;
char INTERACT[130];

main()
{
    printf( "This program generates an application process,\n\0");
    printf( "a number of sublayer processes, and a network\n\0");
    printf( "stub process. Data interactions are passed\n\0");
    printf( "down through all layers and credit interactions\n\0");
    printf( "are passed up in response. The application process\n\0");
    printf( "receives timing interactions from a timer process\n\0");
    printf( "so that it can record the elapsed time. The\n\0");
    printf( "above data cycle is repeated 1000 times with 100\n\0");
    printf( "byte interactions being passed through pipes.\n\0\n\0");

    printf( "    How many sublayers? ...");
    scanf("%d", &n); /* n := number of sublayers */

    pipe( EVENT_PIPE);

    for ( i=n; i>0; i=i-1){
        ID = i;
        pipe( DOWN_PIPE);
        FORK_RESULT = fork();
        if (FORK_RESULT != 0) i=0; /* Parent process exits loop */
        else{ /* Child process' logic */
            UP_PIPE[0] = EVENT_PIPE[0];
            UP_PIPE[1] = EVENT_PIPE[1];
            EVENT_PIPE[0] = DOWN_PIPE[0];
            EVENT_PIPE[1] = DOWN_PIPE[1];
            if (i==0) ID = -1; /* Network ID */
        }
    }

    if (ID == n) applic();
    else
        if (ID == -1) network();
        else
            layer();

    exit(0);
}

```

```

applic()
{
    int ELAPSED_TIME = 0;

    /* Set up timer */
    FORK_RESULT = fork();
    if (FORK_RESULT == 0) { /* Child is timer */
        close( EVENT_PIPE[0]);
        close( DOWN_PIPE[0]);
        close( DOWN_PIPE[1]);
        INTERACT[0] = TIME;
        while(1) { /* do forever */
            sleep(1); /* 1 sec. */
            write( EVENT_PIPE[1], INTERACT, 130);
        }
    }

    /* Parent process' logic */
    close( EVENT_PIPE[1]);
    close( DOWN_PIPE[0]);

    for (i=1; i<=1000; i=i+1){
        INTERACT[0] = DATA;
        write( DOWN_PIPE[1], INTERACT, 130);
        read( EVENT_PIPE[0], INTERACT, 130);
        while (INTERACT[0] == TIME){
            ELAPSED_TIME = ELAPSED_TIME + 1;
            read( EVENT_PIPE[0], INTERACT, 130);
        } /* Credit has now been received */
    }

    printf( "Application finished... Time = %d sec.\n", ELAPSED_TIME);
    kill( FORK_RESULT, 9); /* Kill Child */
    return;
}

```

```

layer()
{
    close( UP_PIPE[0]);
    close( EVENT_PIPE[1]);
    close( DOWN_PIPE[0]);

    for (i=1; i<=1000; i=i+1){
        read( EVENT_PIPE[0], INTERACT, 130);
        write( DOWN_PIPE[1], INTERACT, 130);
        read( EVENT_PIPE[0], INTERACT, 130);
        write( UP_PIPE[1], INTERACT, 130);
    }

    return;
}

```

```
network()
{
    close( UP_PIPE[0]);
    close( EVENT_PIPE[1]);

    for (i=1; i<=1000; i=i+1){
        read( EVENT_PIPE[0], INTERACT, 130);
        INTERACT[0] = CREDIT;
        write( UP_PIPE[1], INTERACT, 130);
    }

    return;
}
```



```

/* Per-process Global Variables */

#define DATA 0
#define CREDIT 1
#define TIME 2

int n, EVENT_PIPE[2], UP_PIPE[2], DOWN_PIPE[2], FORK_RESULT;
char *MAILBOX; /* MAILBOX is used to simulate the passing of pointers */
               /* between pseudo-process layers. */

main()
{
    printf("This program generates an application process.\n");
    printf("a number of sublayer pseudo-processes, and a network\n");
    printf("stub process. Data interactions are passed\n");
    printf("down through all layers and credit interactions\n");
    printf("are passed up in response. The application process\n");
    printf("receives timing interactions from a timer process\n");
    printf("so that it can record the elapsed time. The\n");
    printf("above data cycle is repeated 1000 times with 130\n");
    printf("byte interactions being passed through pipes.\n");

    printf("    How many sublayers? ...");
    scanf("%d", &n); /* n := number of sublayers */

    pipe(EVENT_PIPE);
    pipe(DOWN_PIPE);

    if (fork() == 0)
        if (n > 0) teletex();
        else /* Network process */
            UP_PIPE[0] = EVENT_PIPE[0];
            UP_PIPE[1] = EVENT_PIPE[1];
            EVENT_PIPE[0] = DOWN_PIPE[0];
            EVENT_PIPE[1] = DOWN_PIPE[1];
            network();
    }
    else applic();

    exit(0);
}

```

```

applic()
{
    int ELAPSED_TIME = 0;
    char INTERACT[130];
    int i;

    /* Set up timer */
    FORK_RESULT = fork();
    if (FORK_RESULT == 0) { /* Child is timer */
        close( EVENT_PIPE[0] );
        close( DOWN_PIPE[0] );
        close( DOWN_PIPE[1] );
        INTERACT[0] = TIME;
        while(1) { /* do forever */
            sleep(1); /* 1 sec. */
            write( EVENT_PIPE[1], INTERACT, 130);
        }
    }

    /* Parent process' logic */
    close( EVENT_PIPE[1] );
    close( DOWN_PIPE[0] );

    for (i=1; i<=1000; i=i+1){
        INTERACT[0] = DATA;
        write( DOWN_PIPE[1], INTERACT, 130);
        read( EVENT_PIPE[0], INTERACT, 130);
        while (INTERACT[0] == TIME){
            ELAPSED_TIME = ELAPSED_TIME + 1;
            read( EVENT_PIPE[0], INTERACT, 130);
        } /* Credit has now been received */
    }

    printf( "Application finished... Time = %d sec.\n\n", ELAPSED_TIME );
    kill( FORK_RESULT, 9); /* kill Child */
    return;
}

```

22
OF / DE

```

teletex()
{
    /* This process simulates the actions of n pseudo-process layers */
    char INTERACT[130], *INTER_PTR, *set();
    int i,j;

    UP_PIPE[0] = EVENT_PIPE[0];
    UP_PIPE[1] = EVENT_PIPE[1];
    EVENT_PIPE[0] = DOWN_PIPE[0];
    EVENT_PIPE[1] = DOWN_PIPE[1];
    pipe( DOWN_PIPE);

    if (fork() == 0){ /* Child is network process */
        UP_PIPE[0] = EVENT_PIPE[0];
        UP_PIPE[1] = EVENT_PIPE[1];
        EVENT_PIPE[0] = DOWN_PIPE[0];
        EVENT_PIPE[1] = DOWN_PIPE[1];
        network();
    }
    else{ /* Teletex process */
        close( UP_PIPE[0]);
        close( EVENT_PIPE[1]);
        close( DOWN_PIPE[0]);

        for (i=1; i<=1000; i=i+1){

            /* Wait for data and relay it down */
            read( EVENT_PIPE[0], INTERACT, 130);
            put( INTERACT);
            for (j=1; j<=n; j=j+1) n_PI(); /* Simulate n layers */
            INTER_PTR = set();
            write( DOWN_PIPE[1], INTER_PTR, 130);

            /* Wait for credit and relay up */
            read( EVENT_PIPE[0], INTERACT, 130);
            put( INTERACT);
            for (j=1; j<=n; j=j+1) n_PI();
            INTER_PTR = set();
            write( UP_PIPE[1], INTER_PTR, 130);
        }
    }
    return;
}

```

```

n_PI()
{
    char $INTER_PTR, $set();

    INTER_PTR = set();
    put( INTER_PTR);
    return;
}

```

```

put( INTER_PTR)
char $INTER_PTR;
{
    MAILBOX = INTER_PTR;
    return;
}

```

```

char $set()
{
    return( MAILBOX);
}

```

```

network()
{
    char INTERACT[130];
    int i;

    close( UP_PIPE[0]);
    close( EVENT_PIPE[1]);

    for (i=1; i<=1000; i=i+1){
        read( EVENT_PIPE[0], INTERACT, 130);
        INTERACT[0] = CREDIT;
        write( UP_PIPE[1], INTERACT, 130);
    }

    return;
}

```

APPENDIX B

iRMX/86 Performance Test Program

```

MAIN$PROCESSES: do;
    declare TOKEN literally 'selector';
    FOREVER literally 'while 1';
    $include(!!W:INCLUD/NUCLUS.EXT)
    $include(!!W:INCLUD/HIF.EXT)

    /* Global Declarations */
    declare EXCEPTION word;
    TASK$ TOKEN;
    MBX(12) TOKEN;
    ID$MBX TOKEN;
    TIMER$TASK TOKEN;
    CHAR(2) byte;
    M byte;
    I byte;
    TIME$ID literally '0';
    DATA$ID literally '1';
    CREDIT$ID literally '2';

    APPLIC: procedure public;
    declare INTERACT$SEG TOKEN;
    INTERACT$PTR pointer;
    INTERACT based INTERACT$PTR integer;
    TIME byte;
    I byte;
    EXCEPTION word;
    RESPONSE word;
    INTERACT$SEG = RD$CREATE$SEGMENT( 144, @EXCEPTION);
    INTERACT$PTR = BUILD$PTR( INTERACT$SEG, 0);

    TIME = 0;
    do I=1 to 1000;
        INTERACT = DATA$ID;
        call RD$SEND$MESSAGE( MBX(1), INTERACT$SEG, 0, @EXCEPTION);
        INTERACT$SEG = RD$RECEIVE$MESSAGE( MBX(0), 0FFFFH, @RESPONSE, @EXCEPTION);
        INTERACT$PTR = BUILD$PTR( INTERACT$SEG, 0);
        do while INTERACT = TIME$ID;
            TIME = TIME + 1;
            INTERACT$SEG = RD$RECEIVE$MESSAGE( MBX(0), 0FFFFH, @RESPONSE, @EXCEPTION);
            INTERACT$PTR = BUILD$PTR( INTERACT$SEG, 0);
        end;
        if INTERACT <> CREDIT$ID then do;
            call RD$SEND$CO$RESPONSE( 0, 0, @('ERROR', 10, 13), @EXCEPTION);
            call RD$DELETE$JOB( 0, @EXCEPTION);
        end;
    end;

    call RD$DELETE$TASK( TIMER$TASK, @EXCEPTION);
    do I=1 to TIME;
        call RD$SEND$CO$RESPONSE( 0, 0, @('T'), @EXCEPTION);
    end;

    call RD$DELETE$JOB( 0, @EXCEPTION);
end;

```

```

LAYER: procedure reentrant public;
declare INTERACT$SEG TOKEN,
        INTERACT$PTR pointer,
        INTERACT based INTERACT$PTR integer,
        UP$MBX TOKEN,
        MY$MBX TOKEN,
        DOWN$MBX TOKEN,
        EXCEPTION word,
        RESPONSE word;

```

```

UP$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFHH, @RESPONSE, @EXCEPTION);
MY$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFHH, @RESPONSE, @EXCEPTION);
DOWN$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFHH, @RESPONSE, @EXCEPTION);

```

```

do FOREVER;
    INTERACT$SEG = RO$RECEIVE$MESSAGE( MY$MBX, OFFFHH, @RESPONSE, @EXCEPTION);
    call RO$SEND$MESSAGE( DOWN$MBX, INTERACT$SEG, 0, @EXCEPTION);
    INTERACT$SEG = RO$RECEIVE$MESSAGE( MY$MBX, OFFFHH, @RESPONSE, @EXCEPTION);
    call RO$SEND$MESSAGE( UP$MBX, INTERACT$SEG, 0, @EXCEPTION);
end;
end;

```

```

NET: procedure public;
declare INTERACT$SEG TOKEN,
        INTERACT$PTR pointer,
        INTERACT based INTERACT$PTR integer,
        EXCEPTION word,
        RESPONSE word;

```

```

do forever;
    INTERACT$SEG = RO$RECEIVE$MESSAGE(MBX( N+1), OFFFHH, @RESPONSE, @EXCEPTION);
    INTERACT$PTR = BUILD$PTR( INTERACT$SEG, 0);
    INTERACT = CREDIT$ID;
    call RO$SEND$MESSAGE( MBX( N), INTERACT$SEG, 0, @EXCEPTION);
end;
end;

```

```

TIMER: procedure public;
declare TIME$SEG TOKEN,
        TIME$PTR pointer,
        TIME$INTERACT based TIME$PTR integer,
        EXCEPTION word;

```

```

TIME$SEG = RO$CREATE$SEGMENT( 16, @EXCEPTION);
TIME$PTR = BUILD$PTR( TIME$SEG, 0);
TIME$INTERACT = TIME$ID;
do FOREVER;
    call RO$SLEEP( 10, @EXCEPTION);
    call RO$SEND$MESSAGE( MBX(0), TIME$SEG, 0, @EXCEPTION);
end;
end;

```

```

call RQc$SEND$C$RESPONSE( 0, 0, 0( 56,
'This program creates a system of layered processes and',10,13),
$EXCEPTION);
call RQc$SEND$C$RESPONSE( 0, 0, 0( 56,
'mailboxes. One thousand data (130 byte clocks) and ',10,13),
$EXCEPTION);
call RQc$SEND$C$RESPONSE( 0, 0, 0( 56,
'credit interactions are passed through the system for ',10,13),
$EXCEPTION);
call RQc$SEND$C$RESPONSE( 0, 0, 0( 58,
'timing evaluations. 0.1 sec. ticks will be printed.',10,13,10,13),
$EXCEPTION);

call RQc$SEND$C$RESPONSE( @CHAR, 2, 0( 36,
How many sublayers? ... ),$EXCEPTION);

N = CHAR(1) - 48; /* Convert char. to integer */

ID$MBX = RQ$CREATE$MAILBOX( 0, $EXCEPTION);
do I=0 to N+1;
  MBX(I) = RQ$CREATE$MAILBOX( 0, $EXCEPTION);
end;

TASKS = RQ$CREATE$TASK( 210, @APPLIC, 0, 0, 512, 0, $EXCEPTION);
TASKS = RQ$CREATE$TASK( 210, @NET, 0, 0, 512, 0, $EXCEPTION);

do J = 1 to N;
  TASKS = RQ$CREATE$TASK( 210, @PLAYER, 0, 0, 512, 0, $EXCEPTION);
  call RQc$SEND$C$RESPONSE( 0, 0, 0(3, 'M',10,13), $EXCEPTION);
  call RQ$SEND$MESSAGE( ID$MBX, MBX(I-1), 0, $EXCEPTION);
  call RQ$SEND$MESSAGE( ID$MBX, MBX(I), 0, $EXCEPTION);
  call RQ$SEND$MESSAGE( ID$MBX, MBX(I+1), 0, $EXCEPTION);
end;

TIMER$TASK = RQ$CREATE$TASK( 205, @TIMER, 0, 0, 512, 0, $EXCEPTION);
call RQ$DELETE$TASK( 0, $EXCEPTION);

```

end;


```

MAIN$PROCESSES: do:
    declare YOKEN literally 'selector',
    FOREVER literally 'while 1';
#include(:W:INCLUD/NUCLUS.EXT)
#include(:W:INCLUD/MIF.EXT)

/* Global Declarations */
declare EXCEPTION word,
TASK$ TOKEN,
MBX(3) TOKEN,
ID$MBX TOKEN,
TIMER$TASK TOKEN,
CHAR(2) byte,
N byte,
I byte,
TIME$ID literally '0',
DATA$ID literally '1',
CREDIT$ID literally '2';

APPLIC: procedure public:
declare INTERACT$SEC TOKEN,
INTERACT$PTR pointer,
INTERACT bases INTERACT$PTR integer,
TIME byte,
I byte,
EXCEPTION word,
RESPONSE word;
INTERACT$SEC = RO$CREATE$SEGMENT( 144, @EXCEPTION);
INTERACT$PTR = BUILDPTR( INTERACT$SEC, 0);

TIME = 0;
do I=1 to 1000:
    INTERACT = DATA$ID;
    call RO$SEND$MESSAGE( MBX(1), INTERACT$SEC, 0, @EXCEPTION);
    INTERACT$SEC = RO$RECEIVE$MESSAGE( MBX(0), OFFF$H, @RESPONSE, @EXCEPTION);
    INTERACT$PTR = BUILDPTR( INTERACT$SEC, 0);
    do while INTERACT = TIME$ID:
        TIME = TIME + 1;
        INTERACT$SEC = RO$RECEIVE$MESSAGE( MBX(0), OFFF$H, @RESPONSE, @EXCEPTION);
        INTERACT$PTR = BUILDPTR( INTERACT$SEC, 0);
    end;
    if INTERACT = CREDIT$ID then do:
        call RO$SEND$CO$RESPONSE( 0, 0, @('ERROR',10,13), @EXCEPTION);
        call RO$DELETE$JOB( 0, @EXCEPTION);
    end;
end;
call RO$DELETE$TASK( TIMER$TASK, @EXCEPTION);
do I=1 to TIME;
    call RO$SEND$CO$RESPONSE( 0, 0, @('I',1), @EXCEPTION);
end;
call RO$DELETE$JOB( 0, @EXCEPTION);
end;

```

```

LAYER: procedure public;
declare INTERACT$SEC TOKEN,
        INTERACT$PTR pointer,
        INTERACT based INTERACT$PTR integer,
        UP$MBX TOKEN,
        MY$MBX TOKEN,
        DOWN$MBX TOKEN,
        MAIL TOKEN,
        I integer,
        EXCEPTION word,
        RESPONSE word;

PUT: procedure (SEG);
declare SEC TOKEN;
MAIL = SEC;
end;

GET: procedure TOKEN;
return (MAIL);
end;

UP$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFH, @RESPONSE, @EXCEPTION);
MY$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFH, @RESPONSE, @EXCEPTION);
DOWN$MBX = RO$RECEIVE$MESSAGE( ID$MBX, OFFFH, @RESPONSE, @EXCEPTION);

do FOREVER;
    INTERACT$SEC = RO$RECEIVE$MESSAGE( MY$MBX, OFFFH, @RESPONSE, @EXCEPTION);
    call RO$SEND$MESSAGE( DOWN$MBX, INTERACT$SEC, 0, @EXCEPTION);

    /* Simulate N-1 sublayers */
    do I=1 to N-1;
        call PUT( INTERACT$SEC);
        INTERACT$SEC = GET;
    end;

    INTERACT$SEC = RO$RECEIVE$MESSAGE( MY$MBX, OFFFH, @RESPONSE, @EXCEPTION);
    call RO$SEND$MESSAGE( UP$MBX, INTERACT$SEC, 0, @EXCEPTION);
end;
end;

MET: procedure public;
declare INTERACT$SEC TOKEN,
        INTERACT$PTR pointer,
        INTERACT based INTERACT$PTR integer,
        EXCEPTION word,
        RESPONSE word;

do forever;
    INTERACT$SEC = RO$RECEIVE$MESSAGE( MBX(2), OFFFH, @RESPONSE, @EXCEPTION);
    INTERACT$PTR = BUILDPTR( INTERACT$SEC, 0);
    INTERACT = CREDIT$ID;
    call RO$SEND$MESSAGE( MBX(1), INTERACT$SEC, 0, @EXCEPTION);
end;
end;

```

```

TIMER: procedure public;
declare TIMESSEC TOXEN;
        TIMESPTR pointer;
        TIMEINTERACT based TIMESPTR integer;
        EXCEPTION word;

        TIME$SEG = RO$CREATE$SEGMENT( 16, @EXCEPTION);
        TIME$PTR = BUILD$PTR( TIME$SEG, 0);
        TIME$INTERACT = TIME$ID;

do FOREVER;
        call RO$SLEEP( 10, @EXCEPTION);
        call RO$SEND$MESSAGE( MBX(0), TIME$SEG, 0, @EXCEPTION);
end;

call RO$C$SEND$C$RESPONSE( 0, 0, @ 56,
'This program creates a system of pseudo- processes and',10,13),
@EXCEPTION);
call RO$C$SEND$C$RESPONSE( 0, 0, @ 56,
'buffers. One thousand data (130 byte blocks) and', 10,13),
@EXCEPTION);
call RO$C$SEND$C$RESPONSE( 0, 0, @ 56,
'credit interactions are passed through the system for',10,13),
@EXCEPTION);
call RO$C$SEND$C$RESPONSE( 0, 0, @ 58,
'timing evaluations. 0.1 sec. ticks will be printed.',10,13,10,13),
@EXCEPTION);

call RO$C$SEND$C$RESPONSE( @CHAR, 2, @ 36,
How many sublayers? ... ),@EXCEPTION);

N = CHAR(1) - 48; /* Convert char. to integer */

ID$MBX = RO$CREATE$MAILBOX( 0, @EXCEPTION);

do I=0 to 2;
        MBX(I) = RO$CREATE$MAILBOX( 0, @EXCEPTION);
end;

TASKS = RO$CREATE$TASK( 210, @APPLIC, 0, 0, 512, 0, @EXCEPTION);
TASKS = RO$CREATE$TASK( 210, @NET, 0, 0, 512, 0, @EXCEPTION);
TASKS = RO$CREATE$TASK( 210, @LAYER, 0, 0, 512, 0, @EXCEPTION);
call RO$C$SEND$C$RESPONSE( 0, 0, @ 3, 'M',10,13), @EXCEPTION);
call RO$SEND$MESSAGE( ID$MBX, MBX(0), 0, @EXCEPTION);
call RO$SEND$MESSAGE( ID$MBX, MBX(1), 0, @EXCEPTION);
call RO$SEND$MESSAGE( ID$MBX, MBX(2), 0, @EXCEPTION);

TIMER$TASK = RO$CREATE$TASK( 205, @TIMER, 0, 0, 512, 0, @EXCEPTION);
call RO$DELETE$TASK( 0, @EXCEPTION);

end;

```

APPENDIX C

Teletex Transport Layer Timing Test Program

```

/* This program runs the Transport service module in the end-to-end */
/* DATA state. Alternate data and credit interactions are input.    */
/* This cycle is repeated 1000 times for timing evaluations.          */

```

```

char int1[130], int2[130];
int i, result;

main()
{
    int1[0] = 48;
    int1[1] = '0';
    int1[2] = 2;
    int1[3] = 3; /* Set state to data-wait */
    result = T_service( int1, int2 );

```

```

    for (i=1; i<=1000; i=i+1){

        int1[0] = 34; /* T_DATA_REQ */
        int1[1] = 0;
        int1[2] = 0;
        int1[3] = 0;
        result = T_service( int1, int2 );

```

```

        int1[0] = 75; /* M_CREDIT_IND */
        result = T_service( int1, int2 );
    }

```

```

    exit(0);
}

```

```

pr()
{
    /* Output resultant state indication */
    printf("%d\n", int1[0]);
    printf("%d\n", int1[1]);
    printf("%d\n", int1[2]);
    printf("%d\n", int1[3]);
    printf("\n");

    printf("%d\n", int2[0]);
    printf("%d\n", int2[1]);
    printf("%d\n", int2[2]);
    printf("%d\n", int2[3]);
    printf("\n\n");
    return;
}

```

END

1	2	0	6	8	6

FIN