1

CANADIAN THESES ON MICROFICHE

THÈSES CANADIENNES SUR MICROFICHE

National Library of Canada
Collections Development Branch

Canadian Theses on
Microfiche Service

Ottawa, Canada
K1A 0N4

Bibliothèque nationale du Canada
Direction du développement des collections

Service des thèses canadiennes
sur microfiche

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**

Canada

National Library
of Canada

Bibliothèque nationale
du Canada

CANADIAN THESES
ON MICROFICHE

THÈSES CANADIENNES
SUR MICROFICHE

0-315-19922-9

67829

(21)

NAME OF AUTHOR/NOM DE L'AUTEUR WILLIAM A. MALEK

TITLE OF THESIS/TITRE DE LA THÈSE PORTING AND ANALYSIS OF AN
EXISTING X.25 PACKAGE

UNIVERSITY/UNIVERSITÉ CARLETON UNIVERSITY

DEGREE FOR WHICH THESIS WAS PRESENTED/
GRADE POUR LEQUEL CETTE THÈSE FUT PRÉSENTÉE MASTER OF ENGINEERING

YEAR THIS DEGREE CONFERRED/ANNÉE D'OBTENTION DE CE GRADE 1985

NAME OF SUPERVISOR/NOM DU DIRECTEUR DE THÈSE DR. 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, 1985 SIGNED/SIGNÉ W. Malek

PERMANENT ADDRESS/RÉSIDENCE FIXÉE 162 IVY CRES. UNIT #2
OTTAWA, ONT. K1M 1X6

PORTING AND ANALYSIS OF
AN EXISTING X.25 SOFTWARE PACKAGE

by

William Alexander Malek, B.Eng. (EE)

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

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
Ottawa, Ontario
December 1984

Copyright © 1984, William A. Malek

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

"PORTING AND ANALYSIS OF

AN EXISTING X.25 SOFTWARE PACKAGE"

submitted by William Alexander Malek, B.Eng. (EE), in
partial fulfillment of the requirements for the degree of
Master of Engineering.

Thesis Supervisor

Chairman
Department of Systems and
Computer Engineering

Department of Systems and Computer Engineering
Faculty of Engineering
Carleton University
January 1985

ABSTRACT

This thesis describes the process of porting an existing X.25 multitasking software package from its original environment to another target environment. The porting activity is partitioned into two phases: (1) adaptation of the software to the target environment and (2) confidence testing the port for subtle environmental incompatibilities.

Performance analysis of the ported package is used as a method of comparing the two environments. Mechanisms for improving the performance of communication software are later proposed.

An investigation of the issues impacting the portability of software of this nature is made so that rules for portability enhancement can be derived.

ACKNOWLEDGEMENTS

First and foremost, I wish to express my sincere thanks and appreciation to my thesis supervisor, Dr. R.J.A Buhr, whose continued guidance, support and patience made this thesis a manageable effort.

To the other graduate student members of the original "OSI Gateway" project team, Chris Griffiths and Alex Pepple, I am grateful for several technical consultations, and discussions about the thesis in general.

I wish to thank another fellow graduate student, Spiros Boucouris, for supplying the real-time clock drivers used in the performance analysis section of the thesis.

Also, I thank Dr. B. Pagurek for providing access to the word processor on which the thesis was typed.

I am grateful to my family and friends who showed their support of my efforts with continued interest and encouragement.

Finally, I would like to thank Ingrid Cvar for proof reading this thesis.

TABLE OF CONTENTS

Abstract	111
Acknowledgements	iv
Table of Contents	v
List of Figures	viii
List of Tables	ix
CHAPTER 1 <u>INTRODUCTION</u>	1-1
1.1 Background	1-1
1.2 Objectives	1-1
1.3 Motivation and Goals	1-2
1.4 Current Status	1-4
1.5 Outline of Thesis	1-4
CHAPTER 2 <u>PORTING THE EXISTING X.25 PACKAGE</u>	2-1
2.1 Introduction	2-1
2.2 The Original X.25 Software Package	2-3
2.2.1 Software Structure	2-3
2.2.2 Overview of Software Interactions	2-3
2.2.2.1 The Network Layer	2-3
2.2.2.2 The Link Layer	2-6
2.2.2.3 The Physical Layer	2-7
2.2.2.4 Message Flow Throughout the System	2-8
2.2.2.5 System Status Checking	2-9
2.3 The Original Operating Environment	2-10
2.3.1 The Tradex Kernel and its Interface	2-10
2.3.2 Original Hardware	2-11
2.3.3 Interrupt Processing Mechanisms	2-12

2.4	The Target Environment - iRMX 86	2-14
2.4.1	Why iRMX 86	2-14
2.4.2	iRMX 86 Overview	2-16
2.4.2.1	Nucleus Objects and Interfaces ..	2-16
2.4.2.2	Task Scheduling by the Nucleus ..	2-17
2.4.3	PL/M-80 vs. PL/M-86	2-17
2.4.4	Target Hardware	2-19
2.4.5	Interrupt Processing in iRMX 86	2-20
2.5	Steps Involved in Porting the Software	2-25
2.5.1	Initial Preparations	2-25
2.5.2	Adapting the Software to the New Environment	2-25
2.5.2.1	Adaptation Philosophy	2-25
2.5.2.2	Interfacing to the iRMX 86 Nucleus	2-26
2.5.2.3	Porting to PL/M-86	2-28
2.5.2.4	Changes to the Software/Hardware Interface	2-30
2.5.2.5	Changes to the Interrupt Processing Software	2-30
2.5.2.6	Prioritizing the Tasks	2-33
2.5.3	Results	2-34
2.6	Conclusions	2-35

CHAPTER 3 CONFIDENCE TESTING AND BENCHMARKING THE PORT

		3-1
3.1	Introduction	3-1
3.2	Confidence Testing the Software	3-2
3.2.1	Protocol Requirements	3-2
3.2.2	Test Strategy	3-2
3.2.3	The First Phase	3-4
3.2.3.1	Back-to-back Software Configuration	3-4
3.2.3.2	The Drivers and Stubs	3-5
3.2.3.3	Execution Tracing of Multitasking Software	3-7
3.2.3.4	Problems Revealed by First Test Phase	3-9
3.2.3.5	Test Results	3-14
3.3	Benchmarking the Performance of the Package	3-17
3.3.1	Benchmarking Goals and Strategy	3-17
3.3.2	Analysis of Benchmarked Results	3-18
3.3.2.1	Assertions	3-18
3.3.2.2	Calculations and Results	3-19
3.4	Performance Comparison with the Tradex Environment	3-31
3.5	Conclusions	3-35

CHAPTER 4 PORTABILITY AND PERFORMANCE ISSUES ARISING FROM THE EXERCISE

4-1

4.1	Introduction	4-1
4.2	Factors Affecting Portability	4-2
4.2.1	Introduction	4-2
4.2.2	Operating System Independence	4-3
4.2.2.1	The Generic Approach	4-3
4.2.2.2	Advantages and Examples	4-6
4.2.3	Interrupt and I/O Programming	4-9
4.2.3.1	The Generic Approach	4-9
4.2.3.2	Advantages	4-12
4.2.4	CPU Architecture	4-15
4.2.5	Task Scheduling	4-16
4.3	Improving Performance of X.25-like Software	4-20
4.3.1	Approach	4-20
4.3.2	Theoretical Results	4-21
4.4	Conclusions	4-25

CHAPTER 5 CONCLUSIONS

5-1

REFERENCES

APPENDIX A.	Graphical Notation for Software Structures
APPENDIX B.	Gladiator Monitor Scheduling Algorithms
APPENDIX C.	Skeleton Schematic of Tradex Hardware Configuration
APPENDIX D.	Skeleton Schematic of iRMX 86 Development System Hardware Configuration
APPENDIX E.	Sample Segment of Trace Facility Output
APPENDIX F.	Sample Segment of Benchmarked Output

LIST OF FIGURES

2.1:	Original X.25 Software Structure Diagram	2-4
2.2:	Tradex System Hardware/Software Interactions ...	2-13
2.3:	iRMX 86 OS Layered Subsystems	2-15
2.4:	iRMX 86 Task State Transition Model	2-18
2.5:	iRMX 86 RQ\$WAIT\$INTERRUPT/RQ\$SIGNAL\$INTERRUPT Software Handshake	2-22
2.6:	iRMX 86 Hardware/Software Interactions	2-23
2.7:	Ported Interrupt Handling Structure	2-32
3.1:	Phases of Confidence Testing the Ported X.25 Package	3-3
3.2:	Transport Layer Drivers for the X.25 Package ...	3-6
3.3:	Physical Layer Emulation for the X.25 Package in Back-to-back Configuration	3-8
3.4:	Sequence of Events Leading to Deadlock in the Packet Monitor	3-12
3.5:	Communications Patterns Observed During the First Test Phase	3-16
3.6:	Graphical Representation of Buffer Movement and Processing Times During Full-duplex Data Transfer Phase of the Protocol	3-21
3.7:	Extrapolation of CRC Calculation Time	3-23
4.1:	Generic OS Shell Interface	4-4
4.2:	Generic CREATE\$TASK Primitive	4-7
4.3:	Embedded OS Call Tracing	4-8
4.4:	Emulating the iRMX 86 Itask/ISR Interaction	4-10
4.5:	Generic Implementation of Itask/ISR Interaction	4-13
4.6:	Testing with Hardware Emulation Task	4-14
4.7:	Anomolous Behaviour of Tasks Interacting with Mailbox Pairs in a Timesliced Environment	4-18

CHAPTER 2

PORTING THE EXISTING X.25 PACKAGE

2.1 Introduction

This chapter describes how the original X.25 package was ported from the Tradex environment to the iRMX 86 environment.

Section 2.2 gives an overview of the original X.25 package with respect to its software structure and multitasking interactions. The discussion deals, in particular, with layer interactions, intertask communication methods, and message flow throughout the system.

Section 2.3. briefly outlines the features of the original Tradex operating environment in terms of its kernel, interrupt mechanisms, and the hardware necessary to support the package.

An overview of the development configuration of the target iRMX 86 operating system is given in section 2.4. The versatile features of iRMX 86, in conjunction with the PL/M-86 language, are shown to be well suited as an environment for the communications software package.

CHAPTER 1

INTRODUCTION

1.1 Background

There is currently a trend toward the design of portable, or re-useable software. In general, this trend is motivated by the need to reduce software development costs, which are being forced upward due to an increase in the range of target environments for commercial applications.

In the field of computer communications, the strongest motivation for portability arises from the fact that the software must be supported by at least two systems before it can serve its purpose. Because communications systems are, by definition, distributed, it rarely occurs that multiple target environments are compatible. Consequently, there is a need to identify and investigate the issues related to the portability of protocol software.

1.2 Objectives

The primary objective of this thesis is to investigate by means of a real-life example, the factors that impede protocol software portability. This objective is achieved

by analyzing the original execution environment of an existing X.25 software package. Incompatibilities in the target environment are then resolved and the port is performed. Lessons learned from the porting exercise are studied so that a generic approach for enhancing the portability of communications software can be developed.

The secondary objective of the thesis, is to propose methods of improving the performance of software of this nature. Suggestions made are applied to the results of a benchmark so that we can calculate a theoretical improvement in system throughput.

1.3 Motivation and Goals

The availability of an existing CCITT recommendation X.25 standard software package made this thesis possible. The real-time, multitasking software package, requiring approximately 5000 lines of high-level source code, was developed in the late 1970s at Carleton University. Its original execution environment was developed as an in-house system based on the Intel 8080 microprocessor.

The X.25 software port was originally part of a larger project aimed at forming an OSI Session level gateway between Carleton University's local area network and Datapac. The software was targeted at a single board

implementation of an Intel 8086 microprocessor system running under the iRMX 86 operating system. The board would act as the Datapac interface for a Motorola 68000 based workstation running under Berkeley 4.2 UNIX and supporting the Transport and Session layer protocols. Other graduate student members of the OSI gateway project, under the supervision of Dr. R.J.A Buhr, were responsible for the design, implementation and testing of the Transport and Session layers in the UNIX environment.

The porting exercise was to take place in several phases involving progressive stages of verification of the software/target-environment combination. The original software was available in two varieties: multiple channel and single channel. The simpler single channel version was chosen for the pilot porting attempt in order to gain a quick evaluation of the viability of porting the multiple channel version. Being the analog of a software prototype, the single channel version would serve as a learning tool for general porting practices and also give an estimate of the performance improvement to be expected with the multiple channel version.

Upon verification of a successful pilot port, the techniques developed would aid in porting the multiple channel version to the iRMX 86 environment. The final verified version would be "blown" into ROM, along with the operating system, and executed on a single board computer

containing high-speed, buffered I/O hardware and an interface to the backplane of the UNIX based system.

1.4 Current Status

The initial ambitious goals of the porting exercise were never realized, due to unpredictable failures of the single channel version running in the target environment. Subsequent correspondences with local high-tech organizations also using iRMX 86 have confirmed symptoms of inherent problems with the operating system itself.

Several successful confidence tests and performance benchmarks of the ported package have been made. However, due to the randomness of iRMX 86 system failures, plans to port the multiple channel version of the package have been cancelled. Termination of this portion of the OSI gateway project has also been due, in part, to the emergence of a commercially available X.25 board adaptable to the UNIX systems.

1.5 Outline of Thesis

Chapter 2 gives a brief overview of the characteristics of the X.25 package, and the original and target operating environments. The techniques used to identify and resolve the environmental incompatibilities are described so that

they may be examined in Chapter 4.

In Chapter 3, an outline is given of how the ported package was confidence tested and benchmarked. Subtle operating environment dependencies and target environment failures are explained for the purpose of outlining potential problems with real-time, multitasking systems. The results of the performance benchmark are used for comparison with the original system.

Chapter 4 uses the experience gained in the porting attempt to describe methods of enhancing the portability of real-time, multitasking, communication software. We also approximate the performance improvement that would be expected if some special hardware was added to the target system.

The thesis is concluded in Chapter 5 by relating the achievements to the objectives. An evaluation of what was gained in the porting attempt is given so that possible recommendations and extensions can be discussed.

CHAPTER 2

PORTING THE EXISTING X.25 PACKAGE

2.1 Introduction

This chapter describes how the original X.25 package was ported from the Tradex environment to the iRMX 86 environment.

Section 2.2 gives an overview of the original X.25 package with respect to its software structure and multitasking interactions. The discussion deals, in particular, with layer interactions, intertask communication methods, and message flow throughout the system.

Section 2.3. briefly outlines the features of the original Tradex operating environment in terms of its kernel, interrupt mechanisms, and the hardware necessary to support the package.

An overview of the development configuration of the target iRMX 86 operating system is given in section 2.4. The versatile features of iRMX 86, in conjunction with the PL/M-86 language, are shown to be well suited as an environment for the communications software package.

Section 2.5 describes the steps involved in porting the software from the Tradex environment to the iRMX 86 environment. Specific examples of difficulties encountered during the porting process are illustrated. These examples are to be used in Chapter 4 to support suggestions for enhancing software portability.

2.2 The Original X.25 Software Package

2.2.1 Software Structure

The system software structure diagram, shown in Figure 2.1, is a graphical representation of the original X.25 code written in Intel's iAPX 8080 microprocessor based high level language, PL/M, now known as PL/M-80. The key to structure diagram notation is given in Appendix A.

The code components of the X.25 package consist of a set of tasks interacting in monitors. In this system, each monitor, excluding the BUFFER\$MONITOR, represents one of the three X.25 layers defined in [1]. The software was designed such that the monitors contain the executable code for each X.25 layer, and the tasks provide transportation of data between the monitors.

2.2.2 Overview of Software Interactions

2.2.2.1 The Network Layer

The network layer, in this case, is the X.25 packet level, represented in Figure 2.1 by the PACKET\$MONITOR. It manages a single, flow-controlled virtual circuit and provides five primitive interface procedures to transport layer tasks:

Figure 2.1: Original X.25 Software Structure Diagram

- PLACE\$CALL (CIRCUIT\$ID, REMOTE\$STATION\$ID,
TIMEOUT\$PERIOD, RETURN\$STATUS),
- WAIT\$FOR\$CALL (CIRCUIT\$ID, REMOTE\$STATION\$ID,
TIMEOUT\$PERIOD, RETURN\$STATUS),
- HANGUP (CIRCUIT\$ID, TIMEOUT\$PERIOD,
RETURN\$STATUS,
RETURN\$CALL\$DATA\$VOLUME),
- SEND\$PACKET (CIRCUIT\$ID, PACKET\$DATA,
TIMEOUT\$PERIOD, RETURN\$STATUS),
- RECEIVE\$PACKET (CIRCUIT\$ID, TIMEOUT\$PERIOD,
RETURN\$STATUS).

The first three interface procedures are for call control and the last two for data transfer. Success or failure (including timeout) of the procedure call is reported via RETURN\$STATUS.

On the link layer side of the packet monitor, two interface procedures, PF\$PUT and FP\$GET, provide access for two transport tasks. These tasks, called FP\$PROCESS and PF\$PROCESS, transport packets to and from the packet monitor respectively.

While tasks are inside the packet monitor, they execute the X.25 packet level protocol, as defined in [2]. The packet monitor, in conjunction with its two transport tasks, forms a single Network Entity, as defined in [1], by providing services to the transport layer above it and by requesting services from the link layer below it.

The packet monitor is of the Gladiator Monitor type [3] which enforces task state transitions according to the

scheduling algorithms given in Appendix B.

2.2.2.2 The Link Layer

The link layer in this case is the X.25 frame level protocol. The data structures and executable code associated with the frame level are contained within the FRAME\$MONITOR shown in Figure 2.1. The control procedures of the frame monitor are based on the International Standards Organization (ISO) High-level Data Link Control (HDLC) Link Access Procedure (LAP). It provides a variable-delay, error-free, data-transparent path for packets exchanged over the serial link connecting user equipment to the node.

PF\$PROCESS and FP\$PROCESS transport packets to and from the frame monitor via the interface procedures PF\$PUT and FP\$GET respectively. Being logically part of the packet layer, FP\$PROCESS is responsible for priming the frame monitor with empty buffers into which incoming frames can be copied. This is done via the procedure GIVE\$EMPTY.

On the physical layer side of the frame monitor, PF\$PROCESS and FI\$PROCESS transport frames to and from the frame monitor. A secondary responsibility of these two tasks is to calculate the Cyclic Redundancy Check (CRC) sequence for all frames in transit between nodes.

Task scheduling, within the frame monitor, is also performed according to the Gladiator Monitor algorithm.

2.2.2.3 The Physical Layer

The monitor managing the Physical Layer, called SYNCHRONOUS\$BSC\$ISR (ISR), contains the FSMs and I/O drivers for the physical-link hardware. As indicated by its name, the ISR monitor operates on character-oriented, bit synchronous communication.

Transmission of frames is based on a credit mechanism, whereby FI\$PROCESS must wait on the TRANSMISSION\$COMPLETE semaphore before calling TRANSMISSION\$SEND. The ISR module queues up to three outgoing frames and signals the TRANSMISSION\$COMPLETE semaphore whenever room becomes available in its outgoing queue. This achieves local flow control such that the link layer cannot overload the ISR monitor with too many outgoing frames at any given time.

On the incoming side, IF\$PROCESS calls IF\$PUT either to deposit received frames (if they pass the CRC), or to request empty buffers used for priming the ISR monitor. This is accomplished in the same manner that FP\$PROCESS primes the frame monitor. Priming the ISR monitor with empty buffers for reception of frames is done by calling RECEPTION\$SEND. IF\$PROCESS waits on the RECEPTION\$COMPLETE

semaphore until it is signalled by the ISR that a complete frame has arrived from the physical link. A call is then made to RECEPTION\$GET to obtain the frame.

The three Interrupt Service Routines, which access the I/O ports, are contained within the ISR monitor. RECEPTION\$ISR and TRANSMISSION\$ISR manage communication between the I/O hardware and the link layer. SIGNAL\$ISR performs task switching on a timesliced basis and also signals the TIMER semaphore at regular intervals for protocol timeouts.

The ISR module is actually only a pseudo-monitor, because mutual exclusion on its internal data structures is obtained by interrupt lockout, rather than from the typical scheduling algorithms used by most monitor structures.

2.2.2.4 Message Flow Throughout the System

A pool of free buffers managed, by the BUFFER\$MONITOR, makes the time-consuming process of data copying unnecessary for parameter passing. Instead, a PDU buffer in transition is passed by pointer, thus allowing the data to stay in one area of memory. Tasks call GET\$FREE to acquire empty buffers for initial data copying (e.g. a transport layer task) and return used buffers to the pool via PUT\$FREE. The only purpose of DEAD\$PACKET\$PROCESS is to wait in the frame

monitor (via DEAD\$PKT\$PICKUP) to return acknowledged packets to the free pool.

2.2.2.5 System Status Checking

In terms of link status checking, CONTROL\$PROCESS waits in the frame monitor, via STATUS\$WAIT, for the link to go up or down. Upon initialization of the frame monitor (done via RE\$INIT), CONTROL\$PROCESS waits for the link to come up, notifies FP\$PROCESS via the semaphore DATA\$PAC\$LINE\$UP and initializes the packet monitor via PKT\$STATUS\$REPORT. If, at any time, during normal operation the link goes down, then CONTROL\$PROCESS wakes up and re-initializes the frame and packet monitors.

Protocol timeouts are done by TIMEOUT\$PROCESS which wakes up from the TIMER semaphore periodically and calls TICK and TOCK. Any timeout condition in a monitor wakes up the waiting task and returns it from its entry procedure with a bad status.

2.3 The Original Operating Environment

2.3.1 The Tradex Kernel and its Interface

Development of the original X.25 software was commenced at Carleton University in 1978 by the Microprocessor Systems Development Laboratory (MSDL) group. At that time they had available the 8080 based, ISIS 80 operating system. ISIS 80 was used as a development system, but not as the execution environment for the X.25 application software.

In the interests of time and space efficiency, a small kernel named "Tradex" was tailor-made for simple multitasking applications. It was designed as an environment along with which the application software was downloaded and executed. The features offered by the Tradex kernel included:

- multitasking capability with timesliced task switching,
- real-time interrupt processing,
- SIGNAL and WAIT primitives for semaphores,
- a BLOCK\$AND\$SIGNAL primitive, which allowed a task to simultaneously wait on its private semaphore while signalling another semaphore,
- an UNBLOCK primitive, which allowed a task to unblock another task from its private semaphore,
- ATTACH and DETACH primitives for linked lists.

Indivisibility of Tradex kernel primitives was maintained by a hardware Test-and-Set (TAS) flag and access to the primitives was obtained through a single interface procedure of the format:

```
CALL KERNEL (FOUR$DIGIT$HEX$CODE, DATA$POINTER);
```

FOUR\$DIGIT\$HEX\$CODE was formatted as OXYZ where:

X = code for the operation to be performed (i.e. one of - initialize kernel, create task, wait, signal, block and signal, unblock, perform a context switch, attach to linked list, detach from linked list),

YZ = number representing semaphore, linked list or task control block being accessed,

The variable, DATA\$POINTER, pointed to any data being passed to/from the kernel (e.g. for task creation and linked list operations).

2.3.2 Original Hardware

The X.25 application software was executed on an Intel iSBC 80/20 board. In addition to the 8080 CPU, the 80/20 board supported the following standard Intel peripherals:

- 8251 Universal Synchronous Asynchronous Receiver Transmitter (USART),
- two 8253 Programmable Interval Timers (PITs),
- 8259 Programmable Interrupt Controller (PIC).

The USART was used for communications with the network, while the PITs provided baud rate generation and context switch timing. The PIC was required to manage interrupts from all system peripherals. A skeleton schematic of the Tradex hardware interconnection is shown in Appendix C.

2.3.3 Interrupt Processing Mechanisms

In terms of interrupt management at the hardware level, the PIC was utilized in "master" mode. The resultant effect on the software was that it was possible to have an individual ISR for each of the three interrupt levels. Figure 2.2 depicts a sample hardware/software interrupt processing interaction for a USART receive ready interrupt in this configuration.

As described later, the differences between the interrupt handling mechanisms of Tradex and iRMX 86 necessitated a substantial restructuring of the low level software.

Key

- 1, 2: Interrupt propagation in hardware.
- 3: Transfer of interrupt vector.
- 4: Currently running task preempted.
- 5: Interrupt serviced by ISR.
- 6: Preempted task resumed.

Figure 2.2: Tradex Hardware/Software Interactions

2.4 The Target Environment - iRMX 86

2.4.1 Why iRMX 86

iRMX 86 (RMX) is an Intel iAPX 8086 microprocessor based, real-time, multitasking executive which offers a set of powerful features upon which virtually any application system can be built. RMX was chosen as the target environment for this project because of its availability at Carleton University and for its local commercial interest. It was also considered to be technically appropriate because of its versatile multitasking capability and its native high level language, PL/M-86, which is almost completely downward compatible with the X.25 source code written in PL/M-80.

The RMX operating system as a whole, is comprised of the RMX nucleus and several layered subsystems as shown in Figure 2.3. One of these subsystems, the Human Interface, serves as a window through which RMX was used as a development system. The nucleus however, is the focal point of this discussion because of its importance in the porting process.

Figure 2.3: iRMX 86 OS Layered Subsystems

2.4.2 IRMX 86 Overview

2.4.2.1 Nucleus Objects and Interfaces

Being the core of the RMX operating system, the nucleus provides several "object types" from which other subsystems and application systems are constructed. The wide functional variety of these object types is discussed in Intel literature [4]. However, the purpose of this section is to give a brief overview, rather than a tutorial. Our discussion is therefore constrained to the object types that could be applicable to software of the nature of our X.25 package:

- Task: an active program entity that can exist at one of 256 priority levels,
- Job: an environment in which tasks execute,
- Segment: a piece of memory used for data storage,
- Semaphore: an intertask communication entity used by tasks to send signals to each other,
- Mailbox: an intertask communication entity used by tasks to send objects to other tasks.

Several entry points to the RMX nucleus are provided via interface procedures that enable:

- object creation and deletion,
- operations on objects (e.g. signalling a semaphore),
- suspension, resumption and sleeping of tasks,
- status checking of objects,
- interrupt processing.

2.4.2.2 Task Scheduling by the Nucleus

A task running on the RMX nucleus can be in one of several states. The nucleus governs the scheduling of a task, according to its state and its priority, relative to other existing tasks. For this reason, task scheduling is not done by timeslicing. Instead, the criteria for context switching specify that the currently running task continues to do so until it is suspended, deleted, goes to sleep, or a higher priority task becomes ready to run. The finite state machine model for RMX tasks is shown in Figure 2.4.

When porting from a timesliced environment to a non-timesliced environment, the scheduling philosophy becomes an important factor. This issue is worthy of consideration because each environment has its own characteristics that if relied upon, may impede portability. We shall expand upon this discussion in Chapter 4.

2.4.3 PL/M-80 vs. PL/M-86

PL/M-80 was upgraded to PL/M-86 in order to make efficient use of the 8086 microprocessor's architecture. The major differences impacting the portability of existing PL/M-80 code are at the data type level. PL/M-80 has two data types:

Figure 2.4: iRMX 86 Task State Transition Model

- BYTE: an 8 bit, unsigned quantity,
- ADDRESS: a 16 bit unsigned quantity representing data or a pointer to data.

To take advantage of the 8086's 16 bit architecture and multiple memory segment addressing, PL/M-86 offers several additional data types:

- REAL: a signed, floating point number,
- INTEGER: a signed, 16 bit number,
- WORD: an unsigned, 16 bit quantity,
- DWORD: an unsigned, 32 bit quantity,
- SELECTOR: a 16 bit quantity representing the base address of a memory segment,
- POINTER: a 32 bit quantity representing the segment base and offset portions of a memory address.

Compatibility clashes between the two versions of PL/M are due mostly to the requirement for 32 bits to uniquely specify an 8086 memory address. As discussed in later sections, this discrepancy is one example of a high level language being unable to hide the internal architecture of the processor on which it is executing.

2.4.4 Target Hardware

When ported to RMX, the X.25 software required some form of I/O hardware in addition to the CPU board. In the

available configuration of our development environment, the software was targeted at a Matrox clone of Intel's iSBC 86/12 CPU board which supports several on-board peripherals dedicated to RMX nucleus related functions.

For communications with the "outside world" (other than the user's terminal and the disk), the available I/O hardware was a Matrox clone of Intel's iSBC 534 Four Channel Communications Expansion Board. The 534 board supports several USARTs, PITs, and a PIC. The two boards, which are connected across the MULTIBUS backplane, as shown in Appendix D, interact in a master/slave relationship. When used in conjunction with the RMX operating system, the hardware in this configuration requires different programming sequences and necessitates a different interrupt processing software structure than in Tradex.

This hardware configuration was chosen strictly because it was available and was sufficient for software verification. The final version of the ported software could later be easily adapted to a more compact, single board configuration.

2.4.5 Interrupt Processing in iRMX 86

The basic mechanism for interrupt processing in RMX is more versatile than that of Tradex because it consists of an interaction between an ISR and a special interrupt task

(itask). The itask is logically bound to the ISR and its corresponding interrupt level at run time. Because an ISR runs with all interrupts disabled, its RMX dependent operations are restricted to a small subset of the available nucleus calls. One of these calls involves a simple, RMX nucleus supported wait/signal interaction with its corresponding itask, as graphically represented by Figure 2.5 a).

The occurrence of an interrupt vectors control to the ISR which may service the interrupt itself, or optionally signal the itask if other nucleus calls are required. As shown in the hardware/software interaction diagram of Figure 2.6, this interrupt processing philosophy is different than that of Tradex because interrupt servicing responsibility can be shared between the ISR and itask.

An initialization parameter specifies the maximum number of signals that the ISR can make before the interrupt level is automatically disabled. It remains disabled until the itask performs another wait operation. The versatility arises from the fact that this wait/signal interaction lends itself quite easily to single or multiple buffered ISR/itask interactions, as shown in Figures 2.5 b) and c).

The available development hardware configuration required that the PIC on the 534 board be programmed as a

a) Graphical Representation

b) Single Buffered Timing of Handshake

c) Double Buffered Timing of Handshake

Figure 2.5: ARMX 86 RQ\$WAIT\$INTERRUPT/RQ\$SIGNAL\$INTERRUPT
Software Handshake

Key

- 1, 2, 3: Interrupt propagation in hardware.
- 4: Transfer of interrupt vector.
- 5: Currently running task preempted, ISR invoked.
- 6, 7, 8: Optional invocation of interrupt task.
- 9: Resumption of preempted task.

Figure 2.6: iRMX 86 Hardware/Software Interactions

slave to the master PIC on the CPU board. The resultant effect on any software in this environment is that all external interrupts from the slave PIC are vectored to the same ISR. The slave PIC must then be queried by the software to determine which slave interrupt level was raised. This interrupt processing requirement is more awkward than the multi-ISR configuration of Tradex because more intelligence is required on the part of the ISR.

2.5 Steps Involved in Porting the Software

2.5.1 Initial Preparations

The overall porting activity was partitioned into a two-pass attempt; the first-pass resulting from analysis of structural, syntactical and operating environment incompatibilities and the second-pass resulting from analysis of system failures indicated by test results.

Prior to the first-pass porting attempt, it was necessary to analyze the documentation package [5] in order to gain an understanding of the intertask interactions and how they could be adapted to the new environment. Subsequently, the source code was logically pieced together to derive the level of detail depicted in the system structure diagram of Figure 2.1. Further analysis revealed required modifications due to syntactical and operating environment conflicts.

2.5.2 Adapting the Software to the New Environment

2.5.2.1 Adaptation Philosophy

In the interest of minimizing the code changes necessary to port the package to RMX, an attempt was made to leave the basic algorithmic and architectural structure of the software in its original state. When considering the

differences between the original environment and the target environment, attention was focused on five major changes:

- 1) Changing operating system calls,
- 2) Adapting to PL/M-86,
- 3) Recoding the I/O drivers to match the hardware configuration,
- 4) Adapting the interrupt handling software to support the interrupt handling mechanisms of RMX,
- 5) Prioritizing the tasks to reflect their importance for correct system operation in the non-timesliced environment.

2.5.2.2 Interfacing to the iRMX 86 Nucleus

With intent to minimize changes to the source code, it was determined that taking advantage of the wide variety of RMX objects (and their usage) was not in the best interests of the project. Although it was considered that tailoring the software to make full use of the power of RMX would result in tighter, more efficient code, it was not deemed worth the risk of inadvertently introducing new bugs. Other drawbacks would arise from having to rewrite a substantial amount of the code, as well as a decrease in future portability due to heavy dependence on the RMX nucleus. Consequently, it was decided that all calls to the Tradex kernel would be mapped as directly as possible to their RMX nucleus equivalents.

All Tradex kernel calls for semaphore and task creation were replaced by their RMX counterparts and relocated to initialization procedures in their respective PL/M-86 modules. The relocation was performed mainly to enhance program modularity, which was lacking in the original code.

The two types of primitive semaphore operations - SIGNAL and WAIT - were replaced directly with the RMX calls - RQ\$SEND\$UNITS and RQ\$RECEIVE\$UNITS. However, the Tradex kernel operations - BLOCK\$AND\$SIGNAL and UNBLOCK - used in the scheduling procedures of the Gladiator monitors, posed some problems. While the BLOCK\$AND\$SIGNAL call could be replicated in RMX terms, the UNBLOCK call could not, due to its tight coupling with the idiosyncrasies of the Tradex kernel. This made a rewrite of the monitor scheduling procedures necessary. The rewrite resulted in the need for new data structures for the monitor condition variables and a consequent update of all calls to the scheduling procedures from within the monitors.

When attempting to map the Tradex linked list operations - ATTACH and DETACH - to their RMX equivalents, it was discovered that the buffer monitor had no explicit scheduling algorithm. Instead, the critical section of code that managed the free buffer pool was entirely dependent on the indivisible properties of the Tradex ATTACH and DETACH primitives. Also, the buffer monitor interface procedures,

GET\$FREE and PUT\$FREE, were originally placed (seemingly at random) in different PL/M program modules. These deficiencies prompted a complete rewrite of the buffer monitor using RMX semaphores for scheduling.

All code for initializing process control blocks, semaphores, linked lists, and process queues in the Tradex kernel was deleted because RMX either circumvents, or performs, such housekeeping responsibilities automatically.

2.5.2.3 Porting to PL/M-86

The architectural properties of the 8086 had several impacts on PL/M-86 data types, as outlined in section 2.4.3. Concerning this project was the conversion of the PL/M-80 ADDRESS data type to its PL/M-86 counterpart, the POINTER data type. Conflicts arose due to the double meaning of the ADDRESS data type in PL/M-80 (i.e. a 16 bit WORD or a 16 bit POINTER). The ADDRESS data type was used in both contexts in the original code.

Although "ADDRESS" is not a reserved word in PL/M-86, the compiler, in its attempt to enhance downward compatibility with PL/M-80 code, automatically converts all ADDRESS types to PL/M-86 WORDs. Consequently, all occurrences of PL/M-80 ADDRESS typed variables being used as pointers, are rendered ambiguous by the PL/M-86 compiler. This is because an 8086 memory location requires a 32 bit,

PL/M-86 POINTER to be uniquely specified. A global change of ADDRESS variables to POINTER variables would not suffice because some were used in the context of WORD.

The solution, at first, appeared to be as trivial as hunting through the code for ADDRESS typed variables being used as pointers and redeclaring them as POINTER typed variables. However, after doing so, another problem manifested itself. There were several occurrences of arithmetic operations being performed on ADDRESS typed variables, being used as pointers, to advance the pointers through contiguous memory locations. For example, when processing a packet, the packet monitor would access the different bytes of the packet by incrementing a pointer based at the first element of the packet.

The correct approach is to treat the packet as an array or appropriate data structure and access it in a manner that adheres to structured programming language standards. It was, therefore, necessary to write a special procedure that would deceive the PL/M-86 compiler into allowing the illegal operation of adding a constant to a POINTER. Finally, some small syntactical changes were made for referencing pointers in the new environment.

2.5.2.4 Changes to Software/Hardware Interface

The required changes to the hardware driver modules took place in two phases: hardware initialization code and normal system operation code. The first phase involved deleting all the initialization code that was particular to the hardware used by the Tradex kernel (i.e. real-time clock for time-slicing, and TAS flag for indivisible operations). Due to the difference in hardware configuration of the two environments, it was also necessary to redefine the initialization parameters (i.e. baud rate, master/slave PIC configuration, etc.) and recode them.

The operation code was written using PL/M, LITERALLY defined compilation constants (unlike the initialization code, for some reason). Such a sequence would appear:

```
DECLARE usart$data$port LITERALLY 'OEDH';...
...
```

```
OUTPUT (usart$data$port) = next$byte;
```

Changing the port addresses to match the target hardware only required redefining the LITERALLYs.

2.5.2.5 Changes to the Interrupt Processing Software

The changes to the ISR monitor consisted of adding a new ISR and itask and restructuring the existing coded components of the module. As previously described, a slave level interrupt in RMX vectors control to a single ISR which

optionally signals the itask. As shown in Figure 2.7, upon occurrence an I/O interrupt, the new ISR determines the slave interrupt level requiring service and processes it appropriately. In order to minimize service time overhead, the original RECEPTION and TRANSMISSION ISRs were integrated into branches of a CASE statement in the new ISR, instead of configuring them as subordinate procedures.

Because nucleus calls are not required while processing a frame, the ISR does not signal the itask until a complete frame is sent or received. Hence, the itask is invoked on a per-frame basis, rather than on a per-byte basis. Upon invocation, the itask signals the appropriate semaphore to inform the corresponding frame layer task that frame I/O is complete. The itask and ISR in this configuration form a multiple buffered frame processing interaction.

SIGNAL\$ISR was eliminated altogether because its context switch function is performed automatically in RMX and its remaining functions were assumed by the itask. The consequent elimination of the TIMER semaphore meant that TIMEOUT\$PROCESS had to be configured to continually go to sleep, wake up, and call TICK, TOCK.

Figure 2.7: Ported Interrupt Handling Structure

2.5.2.6 Prioritizing the Tasks

The criterion, for determining the priority of a task in a system, is the importance or urgency of the work that it must do in relation to the other tasks. In some cases, there are several tasks of equal importance. Thus, there is no rule specifying that all tasks must have different priorities. The following is a list of criteria, in descending order, for determining the priorities of the tasks in the ported X.25 package:

- 1) Initialization task: This task must be the highest priority in order to create the rest of the tasks without being pre-empted part-way through the initialization sequence.
- 2) Tasks responding to real-time interrupts: This is the itask servicing Rx done and Tx done interrupts. Data reception and transmission is of high priority because it is time-critical.
- 3) Tasks performing protocol control events: These are TIMEOUT\$PROCESS, which must wake up periodically (without delay) to give protocol timeout signals to the monitors, and CONTROL\$PROCESS, which must have a fast response to link failure.
- 4) Tasks executing the X.25 protocol: These are the transport tasks which perform data transfer between the three protocol layers.

- 5) Transport layer tasks: These are the tasks that call the X.25 interface procedures. The rationale for making them the lowest priority is that they only need to be scheduled whenever all the X.25 support tasks are blocked and waiting for X.25 service requests.

2.5.3 Results

After making the required code modifications, the resultant software structure, as a whole, was not substantially different from that of Figure 2.1. These changes would theoretically be sufficient to run the integrated package on an appropriate testbed. The requirements, at this point in the project, were to design such a testbed and analyze the program output for verification of correct system operation.

2.6 Conclusions

This chapter has given an overview of the efforts involved in porting a relatively complex multitasking software package between two operating environments. The major concerns for porting are the target operating system and its support hardware. These factors, in turn, have an impact on the handling of real-time events and the task scheduling and prioritizing philosophy. In our particular example, CPU differences also affected the high level language standards.

The discussion has proposed a step-by-step approach to porting existing software of this nature, but does not purport to give a global approach to such a problem. While analysis of the differences between two operating environments does tend to expose most of the steps that will be needed for a pilot porting attempt, it cannot guarantee immediate success. The design and implementation of an appropriate testbed for the ported package serves to manifest the subtle incompatibilities.

CHAPTER 3

CONFIDENCE TESTING AND BENCHMARKING THE PORT

3.1 Introduction

This chapter describes the techniques used to test and benchmark the performance of the X.25 package after porting to iRMX 86.

Section 3.2 outlines the overall confidence test strategy and details the first phase of testing the ported software. This section uncovers some interesting examples of porting failures due to subtle dependencies on task scheduling philosophy and memory organization. We also take this opportunity to reveal some problems with the RMX environment itself.

Presented in section 3.3 are the benchmarking methods and results used for performance evaluation of the X.25 package running in the iRMX 86 environment. The primary goal of this section is to calculate which system functions are most demanding on CPU time.

In section 3.4, the benchmarked results are used to estimate performance improvement resulting from the port.

3.2 Confidence Testing the Software

3.2.1 Protocol Requirements

Verification of protocol correctness was not a consideration when designing a test bed for the ported X.25 package because the original system had at one time communicated with Datapac. The sole criterion for determining success of the porting exercise would be to once again, establish communication between the software running under the RMX environment and Datapac.

Satisfying this correctness criterion, however, would result from a final testing phase requiring leasing a line into the Datapac network. Before doing so, a test strategy was needed to give a high degree of confidence that the final testing phase would be a success.

3.2.2 Test Strategy

The complete test strategy was divided into three phases. The first testbed would verify the package using a back-to-back communications configuration completely in software, as shown in Figure 3.1 a). This phase would enable verification of the software logic in absence of the intrinsic timing and debugging problems attributed to hardware dependent events.

a) Test Phase #1: Back-to-back X.25 Protocol Layers

b) Test Phase #2: iRMX 86 - iRMX 86 Communication

c) Test Phase #3: iRMX 86 - DATAPAC Communication

Figure 3.1: Phases of Confidence Testing the Ported X.25 Package

Upon successful completion of the first phase, the second phase would test the integration of the high-level software and the low-level hardware and interrupt programming. As depicted in Figure 3.1 b), this would be accomplished by configuring two RMX machines to communicate over an RS-232 link, imposing system failure conditions and verifying correct execution of system recovery mechanisms.

The third and final phase, illustrated by Figure 3.1 c), would involve connecting an RMX machine to a Datapac line and establishing successful communications with another node.

3.2.3 The First Phase

3.2.3.1 Back-to-Back Software Configuration

To test communication software in the absence of hardware dependent interactions is most easily accomplished either by software loopback or the back-to-back configuration shown in Figure 3.1 a). The latter was chosen because it more closely emulates the peer layer independence of a real communication system. It would also be easier to isolate software failures on either side.

In order to facilitate this configuration, a duplication scheme involving the use of the PL/M "LITERALLY" statement was devised. Each module required two

compilations: the first being a normal compilation, and the second with each interface procedure redefined (using LITERALLY) with the last character of its identifier changed to an underscore. For example:

```
DECLARE PF$PUT LITERALLY 'PF$PU_';
```

This scheme allows the creation of two images of the same code without generating multiple reference errors at module link time. Making a change to the code affects both images and consequently, the programmer's effort involved in code modifications need not be duplicated.

Using the duplication scheme, the layers were linked back-to-back so that a call could be placed from the sending side to the receiving side.

3.2.3.2 The Drivers and Stubs

As shown in Figure 3.2, two tasks, TALKER and LISTENER, were written as transport layer simulation tasks to drive the two packet monitors. TALKER task makes appropriate calls to the X.25 interface procedures - PLACE\$CALL, SEND\$PACKET and HANGUP - while LISTENER makes calls to WAIT\$FOR\$CALL and RECEIVE\$PACKET on the other side.

Figure 3.2: Transport Layer Drivers for the X.25 Package

The PATCH\$MODULE was written to emulate a mutual physical layer for the two frame monitors, as shown in Figure 3.3. Since the interface procedures of the patch module on one side are the same as those of the original ISR module, they are directly replaceable by the ISR module for the second test phase.

3.2.3.3 Execution Tracing of Multitasking Software

Unlike sequential programs that run from initiation to completion in a deterministic manner, tasks in a multitasking environment are mutually dependent in terms of timing and scheduling. To trace the execution history of a multitasking job, it is inappropriate to make I/O calls to any output device while more than one task in that job is still active. This rule is derived from the fact that a task making an I/O call may be suspended while waiting for I/O completion. The consequent context switch changes timing characteristics of the system and may inadvertently resolve intrinsic timing faults by delaying synchronization dependent events. The use of a low-level debugger would disrupt system timing altogether.

The alternative tracing approach taken for this project was to dump trace statements into a trace buffer. After a predetermined time delay from job initiation, a previously dormant trace task would wake up, suspend all tasks and dump the contents of the trace buffer to the appropriate output

Figure 3.3: Physical Layer Emulation for the X.25 Package in Back-to-back Configuration

device. All PL/M statements relating to tracing were conditionally compiled, so that removing them would simply require resetting a compilation switch.

Monitoring the behavioural characteristics of the software involved recording the I.D. of the task making the trace call, as well as the information being traced. The following were the areas of interest for tracing:

- 1) Entry and exit of monitor interface procedures,
- 2) Blocking and unblocking of tasks from all semaphores in the system (including those related to monitor condition variables),
- 3) The values of all monitor protocol state variables, immediately prior to monitor exit by a task,
- 4) The contents of the first 25 bytes of packets and frames in transit between monitors,
- 5) The values of pointers to data in transit between the X.25 interface procedures, and the TALKER and LISTENER tasks.

Appendix E is a sample segment of output generated by the trace facility.

3.2.3.4 Problems Revealed by First Test Phase

The initial runs of the package were unsuccessful due to two problems resulting from poor programming practice by the original authors, as well as some RMX system bugs. The

programming related problems were overlooked during the port because of their subtleties.

The first symptom of poor programming practice became apparent when packets, being taken from the packet monitor for transmission by PF\$PROCESS, were shown by the trace facility to contain random information. A closer trace revealed that packets being queued for transmission by TALKER were somehow being lost in their queue or were being dequeued incorrectly by PF\$PROCESS.

The cause of the failure was proven to be in the data declaration and usage of the internal packet queues in the original packet monitor code. The programmer had relied on contiguous storage of queue data in main memory and the queueing and dequeuing procedures were a reflection of this fact. While this was acceptable in the Tradex system, it became a failing point when compiling onto an 8086 processor because data may be distributed among many memory segments. Redefining the queues and rewriting the queueing procedures rectified the problem.

The second symptom of poor programming practice appeared after correcting the first. Incoming packet level PDUs, when deposited in the packet monitor by FP\$PROCESS, did not awaken the waiting transport layer task, TALKER. An illegal monitor entry by FP\$PROCESS was determined to be the

cause. The entry was termed illegal because the entering process did not call the monitor scheduling procedures, ENTER and EXIT, to acquire and relinquish control of the monitor code.

As illustrated in Figure 3.4, the effect on TALKER, upon being awakened by FP\$PROCESS, was that it would move to the high priority semaphore, but be suspended there because FP\$PROCESS would not call EXIT. Referring to the state transition diagram for Gladiator Monitors (Appendix B), it can be seen that a task suspended on the high priority semaphore is not released until some other task calls EXIT. In this case, PF\$PROCESS was also inside the monitor, suspended on a condition variable semaphore, waiting for a request from TALKER. The result was deadlock.

The reason that the software had executed "correctly" in the TRADEX environment was that TIMEOUT\$PROCESS was calling TOCK at regular intervals, therefore, inadvertently releasing any task suspended on the high priority semaphore. This original programming error was discovered only because the first phase of testing did not activate TIMEOUT\$PROCESS because of its hardware dependence.

We have no conclusive evidence that the illegal monitor entry was an intentional move to eliminate the possibility of FP\$PROCESS being suspended at the packet monitor gate for excessive periods of time. Such waiting would increase the

Key

- 1: Blocked on condition variable, waiting for packet to send.
- 2: Request packet level acknowledgement.
- 3: Blocked on condition variable, waiting for acknowledgement.
- 4: Enter packet monitor without calling "ENTER".
- 5: Deposit packet level acknowledgement and awaken waiting task.
- 6: Leave packet monitor without calling "EXIT".
- 7: Wake up from condition variable.
- 8: Proceed to wait on high priority semaphore --> DEADLOCK.

Figure 3.4: Sequence of Events Leading to Deadlock in the Packet Monitor

chance of incoming frames being bottlenecked at the frame layer because FP\$PROCESS could not dequeue them in time. At any rate, the issue of relying on periodic events (i.e. timeout signalling) for resolving deadlock conditions is analagous to the dependence on timeslicing for correct system operation. This will be a topic for discussion in Chapter 4.

Several problems with the RMX system resulted in loss of confidence in the package/system combination. Consecutive runs of the software on its testbed would manifest different behavioural characteristics. The only traceable problem was that the status variables, being passed by pointer across the X.25 interface procedures, were being lost because their pointer values were somehow being changed upon return. Analysis of the code and experimentation with making all the interface procedures reentrant (which was technically unnecessary) gave no leads. Finally, obtaining a revised version of the PL/M-86 compiler solved the problem.

Rectification of the compiler bug resulted in several successful tests. However, an element of non-deterministic behaviour of the software persisted, as manifested by frequent, unexplainable, untraceable system failures. A Repeated Systems Acceptance Test (RSAT), supplied by Intel, indicated the existence of several RMX hardware-dependent timing bugs. At this point, it was concluded that the

second and third testing phases were not feasible until a reliable RMX system became available. Subsequent events have shown widespread problems among other RMX users.

3.2.3.5 Test Results

The first test phase was designed to determine which X.25 functions the package could perform correctly, rather than those it could not. Consequently, the output from the successful runs indicated the behaviour of the software strictly under normal system operation conditions. Anomolous events, such as lost frames and data corruption, were intended to be imposed on the system during the second test phase. Hence, the software could not be evaluated in this context. The following is a list of software functions that were verified:

- 1) Link start up and initialization (Frame Level),
- 2) Channel restart (Packet Level),
- 3) Call connection on the channel (Packet Level),
- 4) Proper transfer and acknowledgement of several data packets from TALKER to LISTENER (Packet Level),
- 5) Opening and closing of the transmission window for data packets (Packet Level),
- 6) Call clearing on the channel (Packet Level),
- 7) Proper Frame Level acknowledgements for all Packet Level PDU's (Frame Level).

Figure 3.5 illustrates typical communications patterns observed during the first test phase.

3.3 Benchmarking the Performance of the Package

3.3.1 Benchmark Goals and Strategy

Based on the assumption that a reliable RMX system would eventually be available, it was deemed worthwhile to make estimates of the performance of the package in its new environment. Areas of interest included percentage of CPU time devoted to each protocol layer, maximum throughput and bottlenecks.

Times required by individual tasks to perform their duties were logged with the aid of subroutines to utilize the real-time clock on the 534 board. Benchmark trace statements were conditionally compiled in strategic locations throughout the X.25 code for easy removal.

A typical segment of benchmarked output is shown in Appendix F. The comments and data extractions beside the output were inserted manually and account for the overhead involved in servicing the real-time clock. Hand analysis was required because post-processing the data automatically would require a program almost as complex as the X.25 package itself. The complexity arises from the requirement to simulate all semaphore queues and Task Control Blocks (TCBs) in the system.

It is important to recognize that protocol control functions such as link set up, channel restart and call connection are not of significant concern in this discussion because they represent a very small fraction of the overall communication time. Analysis of the data transfer phase of the protocol gives the most realistic, quantitative approximation of system performance.

3.3.2 Analysis of Benchmarked Results

3.3.2.1 Assertions

Analysis of results in this section shall be made on a per-layer basis, to gain insight into which protocol functions are most demanding on CPU resources during the data transfer phase. To achieve this goal, protocol layer responsibilities are segmented according to the boundaries defined in section 2.2.1.

Although the available benchmarked output applies partially to the sending side and partially to the receiving side, we can apply the relevant data wholly to the sending side because the two sides are mirror images. Hence, it is possible to base the calculations on full-duplex communication, even though the benchmarking was performed for half-duplex communication! This in turn allows us to assume full use of acknowledgement piggybacking at both the packet and frame layers.

CPU time allocated to the activities associated with the packet layer is available from the benchmarked data. Benchmarked information regarding frame layer protocol functions is also available, except for CRC generation and checking on PDUs of maximum length. However, it has been indicated by the data that CRC calculation time increases linearly with buffer length. This fact shall be used in the calculations. Estimates must also be made for the physical layer because it was not activated for the first test phase.

In all subsequent calculations, reference to "total" CPU time shall be made. The context of "total" is to be understood as a per packet layer data PDU transmission and acknowledgement. In other words, this is the sequence of events required for a packet layer transmission and reception, with piggybacked acknowledgements. This peer layer interaction advances the transmission window such that the sequence can be repeated for the next outgoing packet layer PDU.

3.3.2.2 Calculations and Results

The following calculations can serve only as a close approximation of system performance, due to the number of assumptions and estimations that will be required. However, wherever possible, three significant figures will be used in the calculations, although results are not to be construed as being accurate to this extent.

Because information is passed by pointer (which precludes data copying), there is no concern regarding buffer length, except at the CRC level or at the physical layer. Figure 3.6 graphically represents the buffer movement and processing times of concern during a full-duplex data transfer phase with acknowledgement piggybacking. Table 3.1 explains the meaning of the annotated transfer times and identifies the source of the data.

Referring to Figure 3.6, total CPU time allocated to the packet layer is defined as:

$$\begin{aligned} t_{\text{packet}} &= t_{\text{pf}} + t_{\text{fp}} \\ &= 32.3 \text{ msec} \end{aligned} \quad (3.1)$$

Similarly, CPU time allocated to the frame layer is expressed as:

$$t_{\text{frame}} = t_{\text{fi}} + (2 \times t_{\text{crc}}) + t_{\text{fr}} \quad (3.2)$$

It is sufficient to double the t_{crc} component, as shown in equation (3.2), because CRC generation on outgoing PDUs and CRC checking on incoming PDUs both involve the same amount of processing. In subsequent text we shall use the term "CRC calculation" when referring to either operation.

To estimate system performance under full load, we estimate maximum CRC calculation time by plotting it as a function of PDU length, as shown in Figure 3.7. The deviant

Figure 3.6: Graphical Representation of Buffer Movement and Processing Times During Full-duplex Data Transfer Phase of the Protocol

Variable Name	Meaning	Source of Data
t_{pf}	Time for outgoing data packet to be processed by packet layer and transported to frame layer.	1
t_{fp}	Time for packet layer to process incoming data packet and awaken waiting transport layer task.	1
t_{fi}	Time for outgoing data frame to be processed by frame layer and transported to physical layer, excluding CRC calculation time.	1
t_{if}	Time for frame layer to process incoming data frame (with piggybacked acknowledgement) and leave it available for pickup by packet layer task. This includes time to return acknowledged buffer to free pool, but excludes CRC calculation time.	1
t_{crc}	CRC calculation time (same for generation or checking).	2
$t_{physical}$	Time required for all physical layer activities.	3

Source of Data

- 1) From benchmark.
- 2) From CRC extrapolation of benchmarked data.
- 3) To be calculated.

Table 3.1: Key to Buffer Movement and Processing Times

Figure 3.7: Extrapolation of CRC Calculation Time

points on the plot result from the fact that the CPU is interrupted every 10 msec for system timer updates. It is evident that the CPU must have been interrupted while these particular PDUs were being processed.

Referring to Figure 3.7, CRC calculation time, y , is a function of the number of bytes being processed, x , according to the equation:

$$y = .0893x + .365 \text{ (msec)} \quad (3.3)$$

For maximum data frame length we get:

$$x = 261 \quad (256 \text{ data, } 5 \text{ overhead})$$

and,

$$y = 23.7 \text{ msec}$$

thus, according to equation (3.2):

$$t_{\text{frame}} = 65.7 \text{ msec}$$

To determine CPU time allocated to the physical layer, we must make several estimations based on information from simple test programs, defined protocol requirements and known operating system characteristics. Table 3.2 explains the variables that we have defined and briefly indicates how their values were determined.

The related effect of these variables on CPU demand at the physical layer is expressed as:

$$t_{\text{physical}} = ((t_{\text{ir}} \times (b_t + b_r))) + (t_{\text{et}} \times b_t) + (t_{\text{er}} \times b_r) + ((t_{\text{cs}} + t_{\text{task}}) \times n_1) \quad (3.4)$$

Note that equation (3.4) does not reference transmission and reception (Tx/Rx) of bytes over the physical link because the USART is essentially another processing entity that executes concurrently with the CPU. Hence, Tx/Rx time at a particular baud rate does not constitute part of the CPU time allocated to the physical layer. It is, instead, considered as a limiting factor when calculating the maximum throughput of the system.

Total CPU time devoted to the physical layer is therefore calculated, according to equation (3.4), as:

$$t_{\text{physical}} = 86.3 \text{ msec}$$

Percentages of CPU time required by each layer can now be calculated:

$$\begin{aligned} \% \text{packet} &= (t_{\text{packet}} / t_{\text{total}}) \times 100 \\ &= 17.5 \% \end{aligned} \quad (3.5)$$

$$\begin{aligned} \% \text{frame} &= (t_{\text{frame}} / t_{\text{total}}) \times 100 \\ &= 35.6 \% \end{aligned} \quad (3.6)$$

and thus,

$$\% \text{physical} = 46.9 \%$$

Our results indicate that the CPU is executing physical layer functions for almost half of total processing time. Frame layer functions are slightly less demanding, being accountable for over one third of total processing time. A breakdown of the respective layer responsibilities reveals some interesting issues regarding their functions.

Note that equation (3.4) does not reference transmission and reception (Tx/Rx) of bytes over the physical link because the USART is essentially another processing entity that executes concurrently with the CPU. Hence, Tx/Rx time at a particular baud rate does not constitute part of the CPU time allocated to the physical layer. It is, instead, considered as a limiting factor when calculating the maximum throughput of the system.

Total CPU time devoted to the physical layer is therefore calculated, according to equation (3.4), as:

$$t_{\text{physical}} = 86.3 \text{ msec}$$

Percentages of CPU time required by each layer can now be calculated:

$$\begin{aligned} \% \text{packet} &= (t_{\text{packet}} / t_{\text{total}}) \times 100 \\ &= 17.5 \% \end{aligned} \quad (3.5)$$

$$\begin{aligned} \% \text{frame} &= (t_{\text{frame}} / t_{\text{total}}) \times 100 \\ &= 35.6 \% \end{aligned} \quad (3.6)$$

and thus,

$$\% \text{physical} = 46.9 \%$$

Our results indicate that the CPU is executing physical layer functions for almost half of total processing time. Frame layer functions are slightly less demanding, being accountable for over one third of total processing time. A breakdown of the respective layer responsibilities reveals some interesting issues regarding their functions.

By looking at equation (3.2) from a different perspective, we can break it down into two components:

$$t_{\text{frame}} = t_{\text{crc}} + t_{\text{fprot}} \quad (3.7)$$

where:

$$\begin{aligned} t_{\text{crc}} &= \text{total time for CRC calculation} \\ &= 47.4 \text{ msec} \end{aligned}$$

$$\begin{aligned} t_{\text{fprot}} &= \text{Time to execute the other frame layer protocol} \\ &\quad \text{functions.} \\ &= 18.3 \text{ msec} \end{aligned}$$

Computing the percentage of t_{frame} used for CRC calculation and for other frame level protocol functions yields:

$$\begin{aligned} \% \text{frame}_{\text{crc}} &= (t_{\text{crc}} / t_{\text{frame}}) \times 100 \\ &= 72.1 \% \end{aligned} \quad (3.8)$$

and,

$$\% \text{frame}_{\text{fprot}} = 27.9 \%$$

In terms of total CPU time:

$$\begin{aligned} \% \text{crc} &= (t_{\text{crc}} / t_{\text{total}}) \times 100 \\ &= 25.7 \% \end{aligned} \quad (3.9)$$

and therefore,

$$\% \text{fprot} = 9.9 \%$$

Thus, over a quarter of total CPU time is required for CRC calculation. By performing a similar breakdown of equation (3.3) for the physical layer, we can calculate how much CPU time is spent for interrupt response and context switch overhead and how much is for actual servicing:

$$t_{\text{physical}} = t_{\text{ircs}} + t_{\text{tser}} \quad (3.10)$$

where,

$$\begin{aligned} t_{\text{tircs}} &= \text{total time for interrupt response and} \\ &\quad \text{context switching at the physical layer} \\ &= (t_{\text{ir}} \times (b_t + b_r)) + (t_{\text{cs}} \times n_1) \\ &= 28.7 \text{ msec} \end{aligned} \quad (3.11)$$

$$\begin{aligned} t_{\text{tser}} &= \text{total time for servicing} \\ &= (t_{\text{et}} \times b_t) + (t_{\text{er}} \times b_r) + (t_{\text{itask}} \times n_1) \\ &= 57.6 \text{ sec} \end{aligned} \quad (3.12)$$

Converting these figures to percentages of CPU time devoted to the physical layer:

$$\begin{aligned} \%_{\text{physical_ircs}} &= (t_{\text{tircs}} / t_{\text{physical}}) \times 100 \\ &= 33.3 \% \end{aligned} \quad (3.13)$$

and thus,

$$\%_{\text{physical_ser}} = 66.7 \%$$

In terms of total CPU time, we get:

$$\begin{aligned} \%_{\text{ircs}} &= (t_{\text{tircs}} / t_{\text{total}}) \times 100 \\ &= 15.6 \% \end{aligned} \quad (3.14)$$

$$\%_{\text{ser}} = 31.3 \%$$

As discussed in Chapter 4, we can reduce, or even eliminate the components of equation (3.10), as well as CRC calculation time. This would allow the allocation of most of the CPU time to the higher level protocol functions. A final summary of our calculations is shown in Table 3.3.

Function performed	Time required (msec)	Percentage of of total CPU time
X.25 protocol for single packet layer data PDU Xmission and acknowledgement	184.3	100 %
Packet level protocol	32.3	17.5 %
Frame level protocol	65.7	35.6 %
Physical level protocol	86.3	46.9 %
CRC calculation	47.4	25.7 %
Other frame level protocol functions	18.3	9.9 %
Interrupt response and itask context switching	28.7	15.6 %
Actual servicing at the physical layer	57.6	31.3 %

Table 3.3: Summary of Benchmark Calculations

Finally, we note that none of the preceding calculations have accounted for CPU time required by PROTOCOL\$TIMER. However, this task actually interacts with all three layers, and thus, the percentages of CPU time required by each layer will remain consistent. For the purposes of our approximation, the overhead imposed by protocol timeout signalling is negligible and does not warrant any further concern.

3.4 Performance Comparison with the Tradex Environment

The most useful comparison that can be made between the ported X.25 package and the original package is that of maximum system throughput.

It has been stated in the original documentation [5] that the software was capable of keeping up with a 9600 baud data transfer rate. It can therefore be concluded that the total CPU processing time to execute the X.25 protocol was less than the time required to communicate with the link at 9600 baud. Otherwise, the physical hardware would intermittently be "waiting" for data, thus defeating the throughput potential of the transfer rate and disrupting the sliding window timing. For our performance comparison, we shall use this assertion to calculate the maximum data transfer rate that could be supported by the ported package.

Total (Tx/Rx) processing time at 9600 baud, for a single packet layer data PDU transmission and acknowledgement, is calculated as:

$$\begin{aligned} t_{txrx} &= (b_t + b_r) \times 8 \text{ (bits/byte)} / \text{baudrate} \\ &= 450 \text{ msec} \end{aligned} \quad (3.15)$$

"Bit stuffing" and deletion functions required for data transparency cannot be taken into account in equation (3.15) due to their non-deterministic properties. However, they can be considered insignificant.

By making a worst-case assumption that t_{txrx} is an upper bound on total CPU processing time in the original system, we can redefine it as:

$$t_{tradex} = 450 \text{ msec}$$

By defining total CPU time in the ported environment as:

$$t_{rmx} = 184.3 \text{ msec}$$

we can calculate the performance ratio:

$$P = t_{tradex} / t_{rmx} \\ = 2.44$$

(3.16)

This speedup factor indicates that we can increase the data transfer rate, the number of logical channels at the packet layer, or a combination of both. Table 3.4 illustrates the theoretical tradeoff between data transfer rate and the number of channels.

Data Transfer Rate (baud)	Number of Channels
1200	19
2400	9
4800	4
9600	2
19200	1

Table 3.4: Theoretical Tradeoff Between Data Transfer Rate and Number of Channels

Reconfiguring the packet layer to handle multiple channels essentially involves using arrays of monitor protocol state variables. This is a theoretical performance tradeoff because adding channels will increase overhead due to the additional number of tasks interacting with the packet monitor. For this reason, there will be a deviation from the theoretical performance as additional channels are implemented.

The performance ratio between the two environments is a rather astonishing revelation, considering that Intel literature [6] claims that the 8086 is seven to ten times more powerful than the 8080. However, it is also stated that processor performance may vary from application to application. One may conclude that the lackluster performance exhibited by the ported software is due to implementation inefficiencies, RMX housekeeping overhead, or a combination of both.

We shall initially postulate that RMX is the cause. According to the benchmarked output, there are typically (non-deterministic) 6 context switches and 20 semaphore wait/signal releases for the sequence of events that have been used for all of our calculations. All context switches were a result of a wait/signal interaction.

Benchmarked output indicates that typical context switch time from a semaphore release is 1.1 msec, and

typical semaphore release time (without context switch) is .65 msec. A simple calculation tells us that RMX housekeeping overhead for these operations consumes approximately 10% of total CPU time. Considering the complexity of RMX, this is not an unreasonable factor and certainly does not account for the low performance ratio.

This revelation directs our suspicions at programming inefficiencies. In this context, we are not referring to poor programming practices, but rather, to sections of code that were inherently better suited to the Tradex environment. However, identifying such characteristics was not the purpose of the benchmark. The discussions in Chapter 4 shall therefore concentrate on improving throughput of the obvious bottlenecks.

3.5 Conclusions

This chapter has described the techniques used to confidence test and benchmark the ported software. The back-to-back configuration of the software on its testbed is an appropriate method for limited confidence testing. However, in circumstances where protocol verification is needed, it does have some intrinsic faults. There are situations in protocols where symmetrical logic errors will cancel themselves in a back-to-back configuration. A simple example of this phenomenon is the "off by one" array indexing error that commonly arises early in the implementation phase of the life cycle. As long as both sides of the configuration are making the same indexing error, the system appears to be working correctly. The discovery of the error is then delayed until communication is attempted with a verified system. An occurrence of this example has been observed in a RMX/PLM implementation of the "COMM" system design found in [7].

Due to the limited testing of this package, we have no verification of the recovery mechanisms for protocol failure. However, given that the system recovery mechanisms were verified in the Tradex environment, we assume that any failure in the RMX environment would be due to subtle hardware dependence. This is not of great concern anymore because the apparent unreliability of our RMX systems has transformed what was originally an ambitious project, into

an academic exercise. The exercise has still been worthwhile because it has allowed us to make an evaluation of the performance of iRMX 86 itself.

It must be stressed that our performance calculations only offer approximate figures. Even when allowing for a large margin of error, we can conclude that the performance ratio between the two environments is rather low when compared with the supposed ratio between the two processors.

CHAPTER 4

PORTABILITY AND PERFORMANCE ISSUES ARISING FROM THE PORTING EXERCISE

4.1 Introduction

In this chapter, we investigate the factors affecting the portability and performance of X.25-like software.

In section 4.2, we discuss the factors affecting portability of communication software and propose a set of general rules that can be used to overcome the target system incompatibilities.

In section 4.3, we use the results of our benchmark to estimate the performance improvement that would be gained by executing bottlenecked functions in hardware.

4.2 Factors Affecting Portability

4.2.1 Introduction

The issue of software portability can be viewed from two levels of target system compatibility; High Level Language (HLL) and operating environment.

HLL incompatibilities are eliminated if the compilers on the target systems adhere to the language standards. The HLL standardization criterion suggests that one would not target the software at systems that do not support the same HLLs. Where such cases are necessary, HLL translators aid the porting activity, but often require some form of human assistance.

HLL portability has been discussed in different contexts [8,9]. However, some more practical issues have been exposed based on the experience gained from the porting exercise described in Chapters 2 and 3.

The discussion in this section concerns the adaptation of software to different operating environments such that minimal changes to the system dependent code are required. With respect to single processor implementations of communication protocols, we partition the characteristics of system dependent software into four categories:

- 1) Low level operating system (OS) primitives typically used for event synchronization, data passing and dynamic allocation of resources and OS entities,
- 2) Real-time and I/O processing mechanisms which differ between operating environments,
- 3) Underlying CPU architectural differences,
- 4) The principle by which pseudo-concurrency is achieved (timesliced or non-timesliced).

4.2.2 Operating System Independence

4.2.2.1 The Generic Approach

To minimize operating system dependence, the application software must be written in such a way as to hide the details of the OS. This is achieved most effectively by creating a generic shell interface to the OS, as shown in Figure 4.1. The interface procedures of the shell are OS primitive look-alikes which make the transformation from OS independent primitives to OS dependent primitives. The embedded code of the shell thus assumes the burden of OS dependence.

Figure 4.1: Generic OS Shell Interface

Our criterion for defining a shell interface procedure as generic is that its internal functions can be emulated via one or more OS primitives, and possibly some additional code, on each target system. The interface characteristics of the shell must not offer any sophisticated features that cannot be emulated on all potential target systems.

An example of a violation of this criterion is an RMX mailbox (used for event synchronization and data passing) which offers a selectable timeout period for tasks waiting for messages. If the timeout period expires before the waiting task receives a message from the mailbox, the task is automatically awakened by the OS and passed an appropriate exception code.

Emulation of the message queueing features of an RMX mailbox in a UNIX environment can be achieved with UNIX pipes (uni-directional message exchanges). However, porting the timeout feature would be awkward because pipes do not facilitate timeouts. To overcome this incompatibility in UNIX would require creating a form of watchdog task to continually monitor the behaviour of the system and take appropriate action in timeout situations.

The key, to making the OS shell generic, is the selection of a minimum subset of OS primitives necessary to support the application software and the mapping of that subset to each of the target operating systems.

4.2.2.2 Advantages and Examples

The generic OS shell offers several advantages over embedding OS calls in the application code. Primarily, the OS adaptation effort involved in porting a software package is as trivial as tailoring the OS shell to the target system.

Another advantage is that parameters for OS shell calls need not match those of the embedded OS primitives. This can be illustrated in the example of the RMX, RQ\$CREATE\$TASK call which requires seven parameters, four of which are not only specific to the iAPX 8086 architecture, but tend to remain fixed during multiple task creations. In addition, exception handling can be done in the shell, rather than in the application software. The generic CREATE\$TASK primitive in the RMX environment, as shown in Figure 4.2 a), needs only two parameters.

For the sake of this argument, let us assume that a PL/M-like language is supported in the UNIX environment. As shown in Figure 4.2 b), we can adapt the generic primitive to the UNIX kernel. The PRIORITY parameter is ignored and the START\$ADDRESS parameter specifies the address of an infinite-loop procedure that would be executed after the "fork" (used for spawning of child tasks).

```

CREATE$TASK:  procedure (PRIORITY, START$ADDRESS) public;
  declare /* Local variables */
  /* Call the actual OS primitive */
  TASK$TOKEN = rq$create$task (PRIORITY,
                                START$ADDRESS,
                                FIXED$DATA$SEG,
                                FIXED$STACK$PTR,
                                FIXED$STACK$SIZE,
                                FIXED$TASK$FLAGS,
                                EXCEPTION$PTR);

  /* Handle the exception */
end CREATE$TASK;

```

a) iRMX 86 Implementation

```

CREATE$TASK:  procedure (PRIORITY, START$ADDRESS) public;
  declare /* Local variables */
  /* Call the actual OS primitive */
  FORK$RESULT = FORK;
  if FORK$RESULT = CHILD$PROCESS then
    call PROCEDURE$AT$START$ADDRESS;
  else
    /* Parent returns from shell */
  end CREATE$TASK;

```

b) UNIX implementation

Figure 4.2: Generic CREATE\$TASK Primitive

Conversely, additional parameters can be added to shell interface procedures for purposes such as tracing behaviour of the software. As shown in Figure 4.3, trace code is embedded in the OS shell. The number of trace calls is greatly reduced because the application software need only pass a procedure or task I.D. as a parameter to the shell, rather than making explicit trace calls preceding each entry to the OS. For a final system, the additional parameters can be ignored upon entry of the OS shell.

```
GENERIC$OS$PRIMITIVE: procedure (... , ... , ... , TASK$ID);  
  ...  
  call TRACE (PRIMITIVE$CALL$ID, TASK$ID);  
  call ACTUAL$OS$PRIMITIVE (... , ... , ...);  
  call TRACE (RESULT$OF$CALL, TASK$ID);  
  ...  
end GENERIC$OS$PRIMITIVE;
```

Figure 4.3: Embedded OS Call Tracing

4.2.3 Interrupt and I/O Programming

4.2.3.1 The Generic Approach

This issue concerns the lowest layer of the protocol and is usually tightly coupled with hardware programming, the characteristics of which are typically governed by the configuration of the I/O, timing and interrupt devices. We are not concerned with I/O port address independence in this discussion because, as described in section 2.5.2.4, literal declaration of compilation constants resolves this problem.

The strategy for making the interrupt and I/O portions of the software portable, is much the same as that for OS independence. Our intent is to hide the interrupt processing mechanisms from the interrupt processing software. The philosophy for selecting a generic set of primitives is to design toward a mechanism that supports the minimum interrupt processing requirements of the most complex system.

To support this argument, we can focus on the RMX interrupt mechanisms and compare with the mechanisms of a hypothetical, Tradex-like system that does not support the explicit binding of an itask to an ISR. Referring to the typical multiple buffering implementation, shown in Figure 4.4, the itask/ISR interaction in RMX is analogous to a

a) iRMX 86 Itask/ISR Interaction

b) Tradex-like System Emulation of Itask/ISR Interaction

Figure 4.4: Emulating the iRMX 86 Itask/ISR Interaction

22
OF / DE

task/ISR interaction via a semaphore and a shared flag in the hypothetical environment.

BUF\$FILLED is a general counting semaphore used by the ISR to signal the task that a buffer has been filled. At initialization time, the count value of the semaphore is zero because no buffers are available. When a buffer has been filled, the ISR signals the semaphore and checks if there are any more empty buffers available. If all the buffers have been filled, the ISR makes a generic call to disable the interrupt level.

The task may be doing other processing while the buffers are being filled and thus, the ISR can continue filling buffers until no more are available. When the task makes a call to WAIT\$INT, it enables the interrupt level just in case it was disabled. This interaction is equivalent to that in RMX because the task must wait if no buffers are ready and the interrupt level is disabled if all the buffers have been filled.

The OS shell in the hypothetical environment hides, from the application software, the fact that the actual OS does not maintain any logical connection between the task and its ISR. In this particular example, the minimum interrupt processing requirements are chosen from the RMX system because all operations can be emulated (or nullified) when ported to the other environment.

4.2.3.2 Advantages

Figure 4.5 a) illustrates how the generic primitives are integrated into the RMX software as a set of procedure calls which may or may not make use of the embedded OS primitives. The internal logic of the itask and ISR consist only of a set of calls to the generic primitives. They, in effect, have no knowledge of their target OS. Porting the module to the hypothetical environment, as depicted in Figure 4.5 b) requires some changes to the generic primitives, but not to the itask or ISR, which are system independent.

A secondary advantage of this scheme is that the logic of the itask and ISR can be tested independent of the hardware. By using a hardware emulation task, as depicted in Figure 4.6, false interrupts can be generated by calling the ISR as a procedure. Both the itask and ISR are unaware of the circumstances leading to their execution and behave as if the events were triggered by hardware.

a) iRMX 86 implementation with generic OS calls

a) Tradex-like system implementation with generic OS calls

Figure 4.5: Generic Implementation of Itask/ISR Interaction

Figure 4.6: Testing with Hardware Emulation Task

4.2.4 CPU Architecture

In our example, the consequence of porting between different CPU environments was twofold:

- 1) The HLL had been upgraded,
- 2) Original programming practices, which relied on the memory structure, caused failures in the ported package.

The ADDRESS data type ambiguity, described in section 2.5.2.3, exemplifies the impact of upgrading a HLL to adapt to a more sophisticated machine architecture. The original programmers could not have foreseen the upgrade and hence cannot be criticized in this context. However, the memory configuration dependent programming practices (i.e. the monitor queueing/dequeueing procedures) were an avoidable cause of trouble. In terms of such practices, we cannot suggest any guides to portability enhancement other than the rules associated with common sense in structured programming.

One of the main purposes of a HLL is to achieve CPU and memory organization independence. Programmers who circumvent the attribute of machine independence are defeating this purpose. The only acceptable reasons for writing sections of machine dependent (i.e. assembly) code are that the function cannot be performed by the HLL or that an increase in speed is required.

4.2.5 Task Scheduling

The issue of porting between multitasking and sequential environments has been investigated [10] and it has been argued that sequential software is the most portable. However, we shall devote this discussion to multitasking issues.

The example of subtle dependence on regular protocol timeout signals for resolving deadlock in the packet monitor, discussed in section 3.2.3.4, raised some important issues. We choose to draw the analogy between this example and task scheduling principles because, in both cases, there is a potential impact on portability.

The way in which pseudo-concurrency is scheduled can have a bearing on the behaviour of the multitasking system. The main principle that we wish to prove here is that the designer, when defining the task interaction structures, must not rely on a timesliced or non-timesliced target environment. Each scheduling philosophy has characteristics that, when ported to the other environment, are susceptible to anomolous behaviour.

Timesliced environments have an intrinsic scheduling property that guarantees the execution of all unblocked tasks within a finite number of timeslice periods. However, there is no guarantee that a system of tasks ported to a

non-timesliced environment will behave in the same manner.

A multitasking program written to rely on timeslicing is susceptible to task starvation situations when ported to a non-timesliced environment. A typical example of this phenomenon is a situation in which a task "hogs" the processor because it never arrives at a blocking condition. Thus, other ready-to-run tasks of equal priority are never scheduled.

Non-timesliced environments have the attribute that race conditions between tasks do not occur. Consider the situation, illustrated in Figure 4.7 a), in which tasks make requests for services at REQ\$MBX and wait for a response status at RESP\$MBX. In a non-timesliced environment, equal priority tasks, TASK\$A and TASK\$B, always receive their responses in the same order as the requests because they proceed from REQ\$MBX to RESP\$MBX without preemption, as shown in Figure 4.7 b). (We must note that tasks of mixed priorities should not interact with the same mailbox pairs because preemption, and a consequent race, is a distinct possibility.)

A potential result of porting this interaction structure to a timesliced environment, as depicted in Figure 4.7 c), is that TASK\$A makes its request at REQ\$MBX and is preempted before proceeding to wait at RESP\$MBX. TASK\$B is

a) Task interaction with mailbox pairs

b) Interaction timing in a non-timesliced environment

c) Interaction timing in a timesliced environment

Figure 4.7: Anomalous Behaviour of Tasks Interacting with Mailbox Pairs in a Timesliced Environment.

then scheduled and proceeds to make its request and wait for its response. When TASK\$A is scheduled to run, it proceeds to wait at RESP\$MBX. Because TASK\$A is behind TASK\$B in the mailbox queue, the responses are not received by their respective requestors.

The rule derived from this example is that designers must aim toward software structures that preclude race or starvation situations in either scheduling environment. In other words, there must be no structural dependence on the presence or absence of regular, periodic events.

4.3 Improving Performance of X.25-like Software

4.3.1 Approach

A common method of increasing throughput of a software bottleneck is to perform its function in high-speed hardware, or eliminate it altogether by performing its function on another processing entity that can run concurrently with the CPU. We shall use our X.25 example to examine the implications of performing bottlenecked functions in hardware.

Bottlenecks in our ported X.25 system can be defined as the software functions that consume the most CPU time and effectively cause other functions to "wait". The software bottlenecks, as determined by the benchmark tests, are at the CRC level and at the physical layer. The selection of appropriate hardware to perform these functions will allow allocation of the CPU completely to the higher levels of the protocol.

Intel produces and documents [11] several 8086-compatible peripherals that perform a variety of communication protocol oriented functions entirely in hardware. For a final single-board configuration there are four peripherals that would be appropriate for our system; the 8273 Programmable HDLC/SDLC Protocol Controller, the 8274 Multi-protocol Serial Controller, the 82530 Serial

Communications Controller, and the 82586 Serial Communications Controller.

All four perform automatic CRC generation and checking, and HDLC framing and synchronization. They are also capable of data transfer rates far in excess of that required by our software. The most versatile of these is the 82586 because it also has four on-chip DMA channels.

It would be a valuable exercise to investigate the result of executing the software on a system configured with an 82586 and dual-ported RAM. The system can be configured for DMA transfers on a per-frame basis because the 82586 requires only the address of memory through which frames are to be exchanged with the CPU. Dual-ported RAM allows the 82586 to work independently of the CPU (without memory bus contention) until a complete frame is processed.

Note that this discussion does not concern the design of the new hardware configuration, but rather investigates the performance implications based on the assumption that such a configuration is feasible.

4.3.2 Theoretical Results

We can predict our performance improvement due to the elimination of the following CPU responsibilities:

- 1) CRC generation and checking,
- 2) Framing and synchronization for data transfer,
- 3) Driving the FSMs that control the physical layer protocol,
- 4) Handling interrupts on a per-byte basis.

Determining the new system throughput involves recalculating total CPU time as defined in section 3.3.2.1 according to the equation:

$$t_{\text{new}} = t_{\text{packet}} + t_{\text{fprot}} + (t_{\text{dma}} \times 2) \quad (4.1)$$

where,

$t_{\text{packet}}, t_{\text{fprot}}$ = As previously calculated,

t_{dma} = Time to service a single DMA interrupt.

For estimation of t_{dma} , we assume that, after initialization, each interrupt is serviced by performing the following functions:

- Sending the 82586 control commands including the new memory pointers for the next set of DMA transfers,
- Signalling the appropriate frame layer semaphore.

Unfortunately, we are not familiar with the programming sequences required to control the 82586. Intel peripherals, in general, have achieved notoriety for difficulty of programming and also for undocumented tricks required to make them work. However, for the purpose of this approximation, we may safely assume an absolute, worst-case

figure of 20 control commands per interrupt. The corresponding time required to service a single DMA interrupt is calculated as:

$$t_{dma} = t_{ir} + t_{cs} + t_{isr} + t_{itask} \quad (4.2)$$

where,

t_{ir} , t_{cs} , t_{itask} = As previously calculated,

t_{isr} = Time to send 20 control bytes to the 82586
 = .04 msec @ 5 MHz

Substituting the values of these parameters yields:

$$t_{dma} = 2 \text{ msec}$$

giving,

$$t_{new} = 54.6 \text{ msec}$$

which results in a new performance ratio of:

$$P_{new} = \frac{t_{tradex}}{t_{new}} = 8.24 \quad (4.3)$$

Thus, by configuring the system for the 82586 and dual-ported RAM, RMX system throughput is further increased by a factor of approximately 3.4. Table 4.1 indicates the theoretical number of channels that could be added to the packet layer with the new hardware configuration.

Data Transfer Rate (baud)	Number of Channels
4800	24
9600	16
19200	8

Table 4.1: Theoretical Tradeoff Between Data Transfer Rate and Number of Channels with the Addition of 82586 and Dual-ported RAM.

Once again, we must be aware of the performance degradation resulting from the overhead of servicing several more transport layer tasks.

4.4 Conclusions

This chapter has proposed methods for enhancing the portability and performance of communications-oriented software. The rules for extended portability have been derived from an analysis of the obstacles overcome during the porting exercise described in Chapters 2 and 3. The overall mechanism, by which system independence is achieved, is the derivation of a generic model of all potential target systems. While it is not possible to predict radical differences between, as of yet, unknown target environments, the generic model should, at least, ease the pain of future ports.

Ada, as a specification language, has been suggested in [7,10,12] because its features (e.g. the rendezvous for intertask communication) are at the same logical level as the language and hence system independent. Implementation can, subsequently, be mapped to the language supported by the target system by using appropriate kernel primitives to emulate these features. However, we maintain our qualification that sophisticated features (e.g. timed entries and accepts) must be avoided because they are not always portable.

The tradeoff for implementing this generic model is a performance degradation that is due to two factors. The first is a direct consequence of the overhead of adding an

extra layer of procedure calls to hide the underlying system. The second results from the fact that intrinsic, time-saving features optimized for the a particular target system may not be utilized. An example of the latter is a set of RMX nucleus primitives that allow the manipulation of sections of shared data. The nucleus automatically protects these sections for mutual exclusion without the overhead of explicit semaphore accesses.

It is up to the designer to decide if the advantages of portability enhancement outweigh the decrease in performance. However, in our opinion, such a time-critical software design would not be a good candidate for portability anyway.

Finally, we have learned that if some dedicated, buffered I/O hardware had been available, much of the effort involved in adapting the existing X.25 interrupt structure to the target hardware could have been avoided.

The resulting, theoretical performance improvement was so great that we can extend this example to a generalization that low-level protocol functions should be performed in hardware. With the new generation of specialized I/O chips becoming available, we believe that there is no justification (other than cost) for performing such functions in software.

CHAPTER 5

CONCLUSIONS

In this thesis, we have analyzed the factors impeding the portability of multitasking, communication protocol software. System dependence has been partitioned into four categories:

- 1) Operating system,
- 2) Interrupt handling mechanisms,
- 3) Underlying CPU,
- 4) Task scheduling principle.

The respective mechanisms for resolving these incompatibilities among different target systems have been:

- 1) Implementation of a generic procedural interface to the operating system,
- 2) Implementation of a generic procedural interface representing the minimum interrupt handling requirements of the most sophisticated target system,
- 3) Strict adherence to high level language programming standards,
- 4) Avoidance, at the design stage, of software structures that are susceptible to the anomalous characteristics of each scheduling philosophy.

It is important that these mechanisms be used a-priori the coding stage, rather than as last-minute patches to a final, non-portable implementation.

We have not made an analysis of the difficulties associated with target systems that do not support the same high level languages. We also recognize that the experience gained in the exercise cannot be generalized to all software structures. However, the criteria for portability need not be confined to our X.25 example.

Performance analysis of our ported software revealed some interesting issues about the two execution environments as well as the software itself. The approximated performance ratio seemed to indicate that the RMX/8086 combination was not as well suited, to the software, as the original Tradex/8080 system.

The temporal breakdown of the individual protocol functions showed that the CPU was spending a disproportionate amount of time performing such mundane activities as CRC calculation and interrupt servicing overhead. This revelation led to an investigation of alternatives methods of handling low-level protocol responsibilities.

The radical performance improvement that would be

expected from the addition of the 82586 communications controller and dual-ported RAM allowed us to conclude that such low-level protocol functions should not be performed in software. The incurred expense of the addition of specialized hardware, however, would be worthy of consideration in low-budget projects.

Although there are not, at this time, any plans to extend the project, the derivation of a generic model of target systems is an important topic for future research. One such project, Advanced Real-Time Tools (ARTT), is currently underway at Carleton University.

REFERENCES

- [1] ISO DP 7498, Reference Model for Open Systems Interconnection, February 1982.
- [2] CCITT Recommendation X.25, Document AP VII-No. 7-E, June 1980.
- [3] Bowen, B.A., Buhr, R.J.A., The Logical Design of Multiple Microprocessor Systems, Prentice-Hall, Englewood Cliffs, N.J., 1980.
- [4] Intel corp., iRMX 86 Nucleus Reference Manual, 1982.
- [5] MSDL Group, The X.25 Package Software Documentation, Edition 1.0, Carleton University, September 1978.
- [6] Intel corp., iAPX 86, 88 User's Manual, July 1981.
- [7] Buhr, R.J.A., System Design With Ada, Prentice-Hall, Englewood Cliffs, N.J., 1980.
- [8] Ford, W.S., Portable Implementation of Network Architecture Layers, ACM, 1983., Ref #. 0-89791-089-3/83/0300-0240.
- [9] Maiden, G.S., Source Translation in a Multi-language Compiling System, Proceedings, ITT Programming, September 1983.
- [10] Griffiths, C.G., Protocol Software Portability, Master's Thesis, Carleton University, December 1983.
- [11] Intel corp., Microsystem Components Handbook, Vol. 2, 1984.
- [12] Buhr, R.J.A., Griffiths, C.G., Organizing Principles for Portable OSI Software, Carleton University, October 1983.

APPENDIX A

Graphical Notation for Software Structures

APPENDIX B

Gladiator Monitor Scheduling Algorithms

LEGEND

→ FORCED TRANSITIONS

→ SELF TRANSITIONS

```

PROCEDURE ENTER:
  BEGIN WAIT (GATE) END

PROCEDURE LEAVE:
  BEGIN
    IF ELIGIBLE.COUNT <> 0 THEN
      BEGIN ELIGIBLE.COUNT:=ELIGIBLE.COUNT-1;
        SIGNAL (ELIGIBLE.SEM) END
    ELSE SIGNAL (GATE)
  END

PROCEDURE SLEEP (VAR C:CONDITION);
  BEGIN
    C.COUNT:=C.COUNT+1;
    IF ELIGIBLE.COUNT <> 0 THEN
      BEGIN
        ELIGIBLE.COUNT:=ELIGIBLE.COUNT-1;
        WAIT-AND-SIGNAL (C.SEM, ELIGIBLE.SEM) END
    ELSE WAIT-AND-SIGNAL (C.SEM, GATE);
    WAIT (ELIGIBLE.SEM)
  END

PROCEDURE AWAKEN (VAR C:CONDITION);
  BEGIN
    IF C.COUNT <> 0 THEN
      BEGIN C.COUNT:=C.COUNT-1;
        ELIGIBLE.COUNT:=ELIGIBLE.COUNT+1;
        SIGNAL (C.SEM)
      END
    END
  END

```

APPENDIX C

Skeleton Schematic of Tradex Hardware Configuration

APPENDIX D

Skeleton Schematic of iRMX 86 Development System
Hardware Configuration

APPENDIX E

Sample Segment of Trace Facility Output

```

*****
link$up = 001, rstate = 000, restart$req = 000, unfit$status (0) = 001,
unfit$status (1) = 000, in$use = 001, request = 000, pf$work = 000,
pstate = 002, c$timer$state = 001, c$timer$count = 127, in$address (0) = 116,
in$address (1) = 000, in$address (2) = 000, in$address (3) = 003,
in$address (4) = 002, in$address (5) = 051, in$address (6) = 001,
in$address (7) = 114, in$address (8) = 000, out$address (0) = 065,
out$address (1) = 004, out$address (2) = 000, out$address (3) = 003,
out$address (4) = 002, out$address (5) = 116, out$address (6) = 002,
out$address (7) = 126, out$address (8) = 016, call$status$block (0) = 001,
call$status$block (1) = 000, ow$timer$state = 000, ow$timer$count = 000,
oh = 000, ot = 000, lwe = 000, nopsn = 000, a$timer$state = 000,
a$timer$count = 000, ih = 000, it = 000, nipsn = 000, xfr$req = 000,
lcst = 000, in$volume = 000, out$volume = 000, dstate = 001,

```

```

*****
pf process, signal pkt$hi$pri, pf got a packet,
009, 250,
11000010, 00010010, 00001010, 00000000, 11000010,
00010011, 00010000, 00000001, 00001011, 01000001,
00000100, 00000000, 00000011, 00000000, 11000010,
10000111, 11000011, 10100000, 11000100, 10100001,
11000101, 10100010, 11000010, 00010111, 11000010,
pf process, wait frame gate, CALL f$awaken, pf process, signal fi work,
pf process, signal frame hi pri, get$free, pf process, wait buff gate, 011,
008, pf process, signal buff gate, CALL pf$get, pf process, wait pkt gate,
fi process, wait frame hi pri

```

```

*****
dce$disc$recd = 000, status = 002, pbc = 002, dce$busy = 000,
timer$recovery = 000, p$bit$sent = 000, odp = 001, easn = 001, vs = 002,
x = 002, s = 002, timer$state = 001, timer = 000, expiry$count = 000,
small$buff$cnt = 001, send$frame$cnt$0 = 002, send$frame$cnt$1 = 001,
send$frame$cnt$2 = 000, send$frame$cnt$3 = 001, send$frame$cnt$4 = 000,
send$frame$cnt$5 = 000, send$frame$cnt$6 = 001, send$frame$cnt$7 = 000,
send$frame$cnt$8 = 006, sac = 002, p$bit$recd = 000, sat = 000,
send$frame = 000, vr = 001, last = 001, eh = 003, et = 004, fh = 001,
ft = 001, rec$frame$count$0 = 001, rec$frame$count$1 = 001,
rec$frame$count$2 = 000, rec$frame$count$3 = 001, rec$frame$count$4 = 000,
rec$frame$count$5 = 000, rec$frame$count$6 = 001, rec$frame$count$7 = 000,
rec$frame$count$8 = 000,

```

APPENDIX F

Sample Segment of Benchmarked Output

103	6708	1636	-----
104	7846	1138	DEAD PACKET PROCESS (S)
31	9133	1287	
2	9662	529	RETURNS ACKED BUFFER TO FREE POOL
3	11525	1862	
30	12040	515	AND WAITS AT FRAME GATE FOR NEXT
100	12573	533	----- DT = 4010 USEC
103	14274	1700	FI PROCESS (S)
104	15414	1139	FALSE AWAKENING, GOES BACK TO SLEEP
102	17381	1966	----- DT = 3314 USEC
101	19016	1635	DEAD PACKET PROCESS (S) ADVANCES INTO FR MON
102	20182	1165	----- DT = 1805 USEC
73	21843	1661	
4	28625	5781	TALKER TASK
5	30475	1849	EXITS SEND*PKT AND CALLS SEND*PKT AGAIN FOR
72	31113	638	NEXT DATA PKT ----- WINDOW
90	31670	556	STILL OPEN
91	32800	1130	----- DT = 9630
93	34765	1964	
94	38118	3352	PF PROCESS (S)
21	40428	2310	
2	40967	538	GETS DATA PACKET
3	42834	1866	
22	43345	511	DEPOSITS IN FRAME MONITOR
100	43959	614	
101	45099	1139	AND GOES BACK TO WAIT FOR NEXT PACKET
23	47512	2413	
4	48049	536	
5	49915	1866	
20	50440	525	
90	50973	533	
91	52111	1137	
92	53342	1231	----- DT = 13072 USEC
103	55045	1702	
104	56183	1137	FI PROCESS (S)
43	57923	1739	
44	58424	501	GETS DATA PACKET
45	60401	1977	
46	60903	501	DEPOSITS IN ISR MODULE
47	64622	3719	
40	65127	505	AND GOES BACK TO WAIT FOR NEXT
41	66266	1138	
42	66768	502	
100	67298	529	
101	68437	1139	
102	69916	1478	----- DT = 10100 USEC

END

2 4 0 2 8 6

FIN