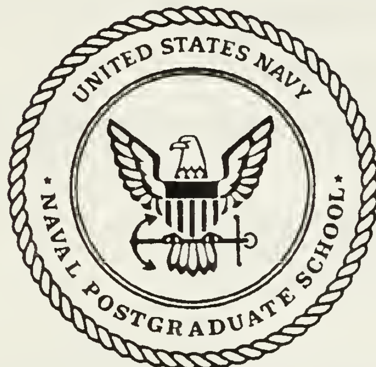
DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY CALIFORNIA 93943-5002

NAVAL POSTGRADUATE SCHOOL
Monterey, California



THESIS

ON PROGRAMMING TRANSPUTERS TO CAPTURE
ADA MULTITASKING FOR THE
NPS AUTONOMOUS UNDERWATER VEHICLE

by

Clay Richmond

December, 1991

Thesis Advisor: Shridhar B. Shukla
Second Reader: Roberto Cristi

Approved for public release; distribution is unlimited.

T258487

Jnclassified

ecurity Classification of this page

REPORT DOCUMENTATION PAGE

1a. Report Security Classification UNCLASSIFIED		1b. Restrictive Markings
2a. Security Classification Authority		3. Distribution Availability of Report Approved for public release; distribution is unlimited.
2b. Declassification/Downgrading Schedule		
4. Performing Organization Report Number(s)		5. Monitoring Organization Report Number(s)
6a. Name of Performing Organization Naval Postgraduate School	6b. Office Symbol *(if applicable)* Code 33	7a. Name of Monitoring Organization Naval Postgraduate School
6c. Address *(City, State, and ZIP Code)* Monterey, CA 93943-5000		7b. Address *(City, State, and ZIP Code)* Monterey, CA 93943-5000
8a. Name of Funding/Sponsoring Organization	8b. Office Symbol *(if applicable)*	9. Procurement Instrument Identification Number
8c. Address *(City, State, and ZIP Code)*		10. Source of Funding Numbers

Program Element Number	Project No.	Task No.	Work Unit Accession No.

11. Title *(Include Security Classification)* ON PROGRAMMING TRANSPUTERS TO CAPTURE ADA MULTITASKSING FOR THE NPS AUTONOMOUS UNDERWATER VEHICLE

12. Personal Author(s) Richmond, Clay A.

13a. Type of Report	13b. Time Covered	14. Date of Report *(Year, Month, Day)*	15. Page Count
Master's Thesis	From ______ To ______.	December 1991	103

16. Supplementary Notation The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the United States Government.

17. Cosati Codes			18. Subject Terms *(Continue on reverse if necessary and identify by block number)*
Field	Group	Subgroup	ADA, Channels in ADA, Multitasking, Processor Communication, Task, Transputer.

19. Abstract *(Continue on reverse if necessary and identify by block number)*

This thesis is in support of the on-going Autonomous Underwater Vehicle (AUV) project at the Naval Postgraduate School in Monterey California. This work investigates the development of a transputer-based multiprocessor and how to program it using Ada.

The objective is to create a software layer that enables intertask communication over a network of transputers to be location invariant and to make the communication process transparent to the user. Ada, being a concurrent language, was chosen as the language in which this software layer is to be written.

The method of intertask communication developed here captures the Ada rendezvous semantics, provides reliable and efficient delivery of messages between tasks regardless of their locations, and uses a common message format for all communicating tasks. The location invariant property makes the software layer particularly suitable for developing higher level allocation algorithms. The communication is handled by generic tasks common to each transputer and a common mapping function that has the locations of all the tasks.

20. Distribution/Availability of Abstract ☒ unclassified/unlimited ☐ same as report ☐ DTIC users		21. Abstract Security Classification UNCLASSIFIED
22a. Name of Responsible Individual Shridhar B. Shukla	22b. Telephone *(Include Area Code)* (408) 646-2764	22c. Office Symbol EC/Sh

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete — security classification of this page

Unclassified

19. [Item 19] Continued:

The programmer needs only to conform to a common format of communication when sending message between tasks and not be concerned with the actual delivery of the message. The software develope was successfully tested and its performance analyzed for a five transputer ring network using the AUV-I data-flow diagram.

Approved for public release; distribution is unlimited.

On Programming Transputers to Capture
Ada Multitasking for the
NPS Autonomous Underwater Vehicle

by

Clay A. Richmond
Lieutenant, United States Navy
B.S., United States Naval Academy, Annapolis 1984

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1991

T2503
R3946
c.1

ABSTRACT

This thesis is in support of the on-going Autonomous Underwater Vehicle (AUV) project at the Naval Postgraduate School in Monterey, California. This work investigates the development of a transputer-based multiprocessor and how to program it using Ada.

The objective is to create a software layer that enables intertask communication over a network of transputers to be location invariant and to make the communication process transparent to the user. Ada, being a concurrent high level language, was chosen as the language in which this software layer is to be written.

The method of intertask communication developed here captures the Ada rendezvous semantics, provides reliable and efficient delivery of messages between tasks regardless of their locations, and uses a common message format for all communicating tasks. The location invariant property makes the software layer particularly suitable for developing higher level allocation algorithms. The communication is handled by generic tasks common to each transputer and a common mapping function that has the locations of all the tasks. The programmer needs only to conform to a common format of communication when sending messages between tasks and not be concerned with the actual delivery of the message. The software developed was successfully tested and its performance analyzed for a five transputer ring network using the AUV-II data-flow diagram.

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

TABLE OF CONTENTS

I. INTRODUCTION 1

A. BACKGROUND 1

B. PROCESSOR AND COMPUTATIONAL REQUIREMENTS 1

1. Timing Requirements 3

2. Proposed Architecture 3

C. OBJECTIVES 4

D. ORGANIZATION 6

II. PROPOSED AUV-II ON-BOARD MULTIPROCESSOR 7

A. TASK RELATIONSHIPS FOR THE AUV-II 7

B. TRANSPUTERS 11

1. Overview 11

2. Transputer Links 11

3. Network Architecture 12

4. Transputer Memory 13

C. PROGRAMMING TRANSPUTERS 14

1. OCCAM 14

2. Interfacing Ada with Transputers 14

III. ADA PROGRAMMING OF A TRANSPUTER NETWORK 16

A. PRIMITIVES OF ADA AND THEIR USE 16

1. Entry/Accept Calls 16

2. Select Statements 17

3. Reading and Writing to Channels 18

4. Delay Statements 19

5. Read_Or_Fail / Write_Or_Fail Statements 19

B. ADA AND ITS USE WITH TRANSPUTERS 19

1. Design Considerations for Ada 20

2. Communications Primitives Available in Ada 20

3. Ada as a DoD Standard 21

C. INTERFACING ADA TO TRANSPUTERS USING OCCAM 21

1. The OCCAM Harness 21

2. Static Allocation 23

IV. BUILDING A COMMUNICATIONS PACKAGE 25

A. OBJECTIVES AND DESIRED BEHAVIOR 25

1. Design Criteria 26

2. Goals 26

B. SOFTWARE COMMUNICATION LAYER 27

1. Concept 27

2. Message Format 31

3. Communication Architecture 32

C. CAPTURING THE RENDEZVOUS SEMANTICS 33

V. PERFORMANCE ANALYSIS 35

A. SIMULATION ARCHITECTURE 35

B. RESULTS 37

C. LIMITS ON PERFORMANCE 40

VI. CONCLUSIONS AND FUTURE WORK 42

A. CONCLUSIONS 42

B. FUTURE WORK 42

1. Higher Level Program 42

2. Difficulties 43

3. Parallelism 44

APPENDIX A: OCCAM SOURCE CODE 45

APPENDIX B: ADA SOURCE CODE 53

APPENDIX C: INVOKE AND LINKING FILES 85

LIST OF REFERENCES 90

INITIAL DISTRIBUTION LIST 92

LIST OF TABLES

Table I : Description by file extension 23

Table II : Task locations 38

Table III : Average iteration times versus task allocations 39

Table IV : Average iteration time versus queue size 40

Table V : Measured iteration times when TIMER frequency is controlled 40

LIST OF FIGURES

Figure 1	:	AUV-II layout	2
Figure 2	:	Block diagram of the T800 transputer [TRANS 89]	4
Figure 3	:	Block diagram of the GESPPU-1 [G64 90]	5
Figure 4	:	Transputer network interface with the host	6
Figure 5	:	Data-flow diagram for the AUV [FLOYD 91]	8
Figure 6	:	Four node transputer network	13
Figure 7	:	Relationship between files for Ada on transputers	22
Figure 8	:	Overall functionality of software layer	25
Figure 9	:	Communication layer structure	27
Figure 10	:	Communication topology	32
Figure 11	:	Message flow at a transputer	35
Figure 12	:	Simulation data-flow	36

ACKNOWLEDGEMENTS

Though this thesis bears my name, there are a number of others who, without whose expertise, effort, and understanding, I could not have completed it. First, I am grateful to John Locke for getting me started on the transputers, answering my endless questions, and making available to me all his previous work. I am also indebted to Uno Kodres for his support and granting me free use of the transputer laboratory. I am extremely thankful to my advisor, Shridhar Shukla, for his patient guidance, insightful ideas, and constant availability. I would like to thank my friend and school partner, Dionysios Makris, who had the unerring ability to point out hidden facts that make engineering, and life, make more sense. He is a true engineer, and his friendship will be with me always.

Lastly, I would like to give my heart felt gratitude to my wife, Kathy, for her love and uncomplaining support, which she gave towards the completion of this thesis, while at the same time coping with problems common to newly expecting mothers. Without her sacrifices, this thesis would not only have been impossible, but also pointless.

I. INTRODUCTION

A. BACKGROUND

The Naval Postgraduate School (NPS) is currently involved in a multi-year project to develop a prototype Autonomous Underwater Vehicle (AUV). This is an interdepartmental project involving the Computer Science, Electrical & Computer Engineering, and Mechanical Engineering departments.

The research for this project was started in 1987 under the sponsorship of the Naval Surface Warfare Center (NSWC) at White Oak, Maryland. Since then, there have been two generations of AUV's. AUV-I was a small vehicle, which could be carried by hand and relied on a radio link for control signals. It was also connected to an umbilical cord for conveying sensor data to the computer and for receiving power. The more recent AUV-II is over four feet long, weighs over 350 pounds and is totally self contained [GOOD 89]. Figure 1 shows the current layout of the AUV-II supplied by the Department of Computer Science.

B. PROCESSOR AND COMPUTATIONAL REQUIREMENTS

The AUV-II uses a GESPAC computer cardcage with a 68030 CPU as its processor. In future models, a parallel processor based on transputers will be used to increase the performance adequately to meet the growing computational requirements. Furthermore, the use of a transputer-based parallel processor is planned to simplify the complex task of the software engineer by being able to modularize the many processes needed for its autonomous behavior.

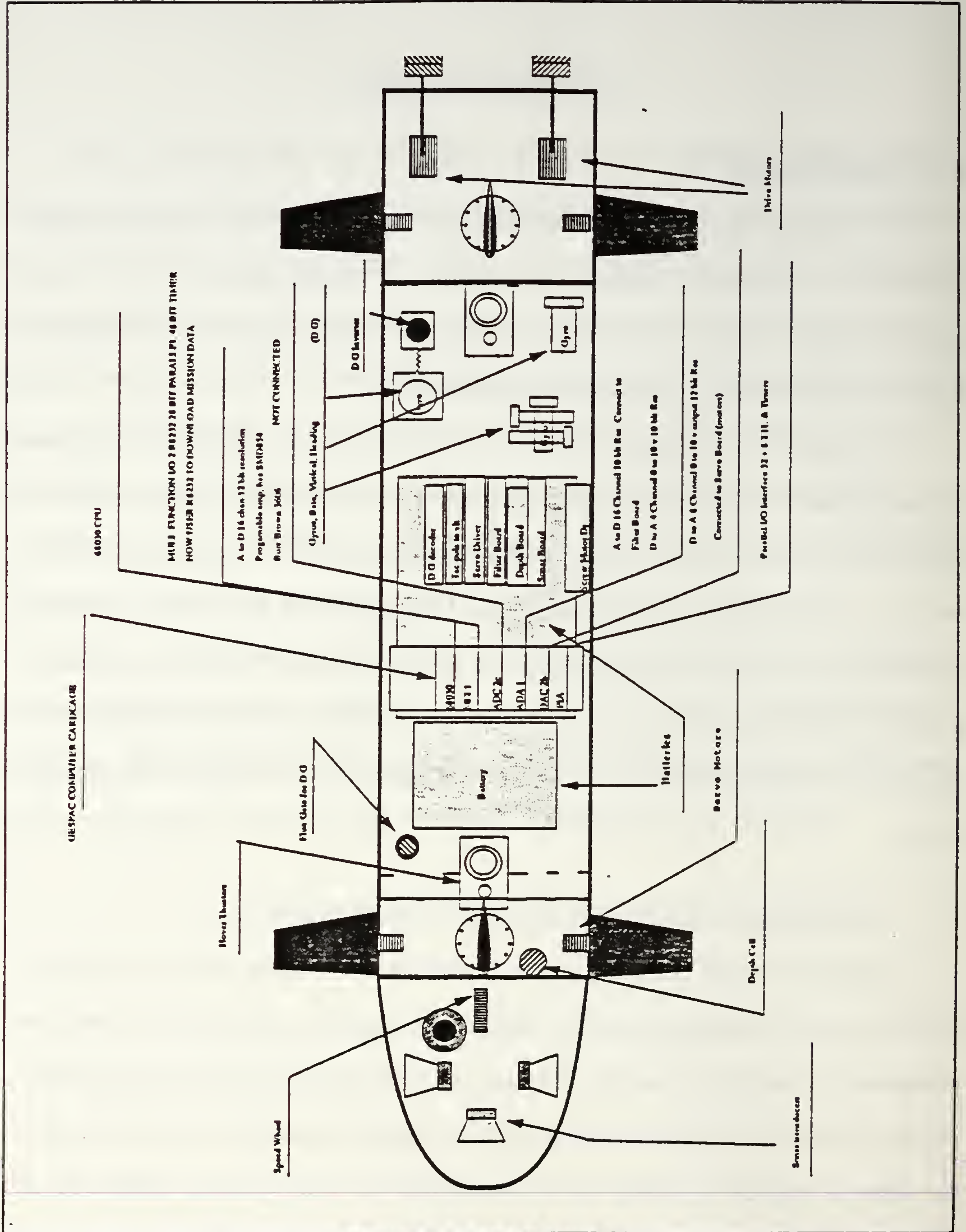


Figure 1 : AUV-II layout

1. Timing Requirements

The AUV-II operates in a real-time environment that requires certain time deadlines to be met [CLOUTIER 90]. When using a single processor, meeting deadlines imposed by the application for both periodic and aperiodic processes is a critical problem. Allocation of processor time so that all the timing requirements are met is of the utmost importance [MAKRIS 91]. As the onboard computer is burdened with carrying out more and more processing to support the intelligent behavior of the vehicle, a single processor is unlikely to be able to meet all the timing requirements. By using multiple processors, the throughput can be increased to meet all the requirements.

The desired frequency of execution of each process for running the AUV-II is 10 hz (with the exception of the sonar, in which case it is likely to be higher). As the vehicle becomes more intelligent, the amount of processing to be done at this frequency will increase.

2. Proposed Architecture

A transputer is a microcomputer that is especially designed to communicate via links to other transputers. It has its own local memory and provides the interfaces for each of the communication links [INMOS REF 86]. Figure 2 shows the block diagram of the T800 transputer.

A possible interface to incorporate transputers is the GESPPU-1/GESPPU-2 combination [G64 90]. Advantages of these cards are that they may be used with a 68030 or IBM PC as hosts, and that any future additions of processors would require only the plugging in of another GESPPU-2 card [GESPAC 90].

The GESPPU-1 (see Figure 3) has one transputer (T800) and provides the interface between a transputer network and the host. One or more GESPPU-2 cards, each

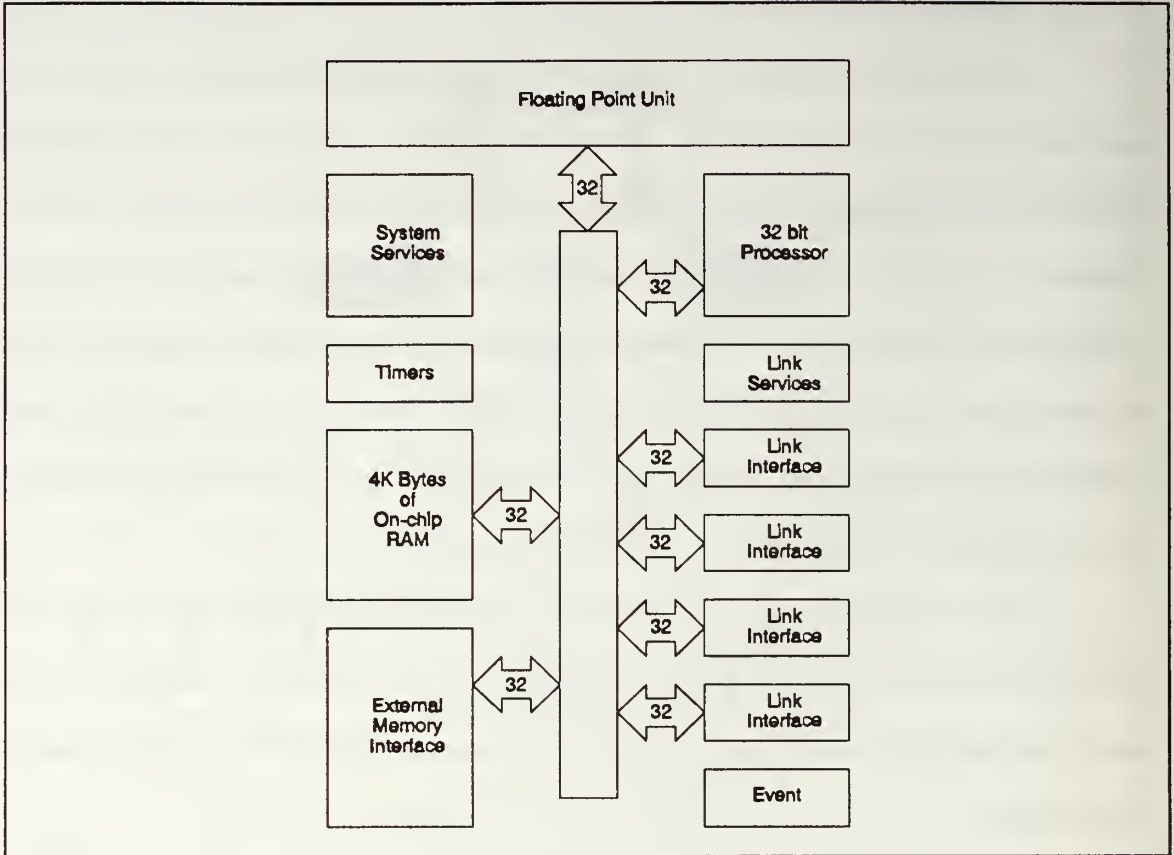


Figure 2 : Block diagram of the T800 transputer [TRANS 89]

containing two transputers (T800's), can be added to construct a network of the desired size [GESPAC 90]. Figure 4 shows the block diagram of the network interface with the host.

C. OBJECTIVES

The objective of this thesis is to use a concurrent programming language on a set of processors. In particular, the goal is to create a software layer to enable the programmer to write a single program and run it on a network of transputers. This would keep the rigorous details of interprocessor communication, such as message passing protocols and synchronization, away from the programmer and give the illusion of a single program

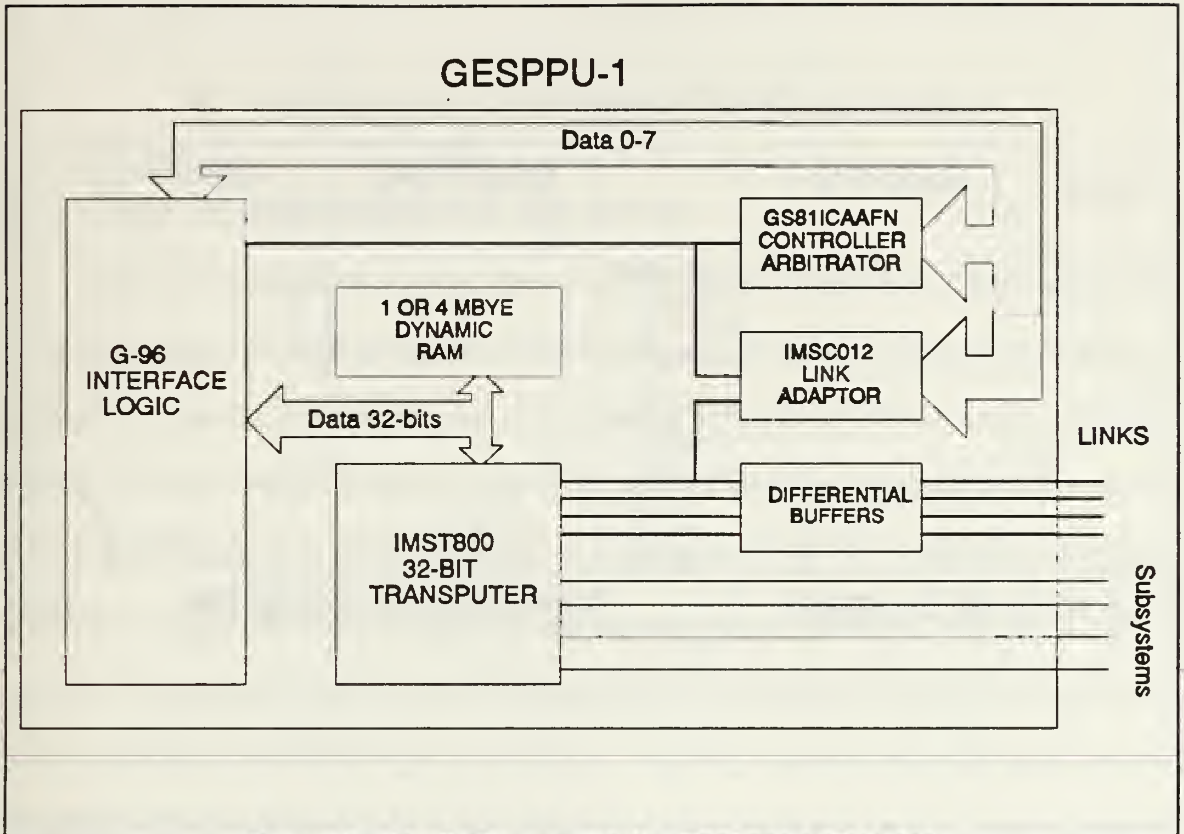


Figure 3 : Block diagram of the GESPPU-1 [G64 90]

running on multiple processors.

The premise of this software layer is to be able to handle all the communication necessary between all the tasks that are running on the network. This would make it possible for a software designer to construct a task oriented program and be able to run that program on one or more processors without regard to the location of the individual tasks and the inherent communications needed. Thus, this communication layer makes the network transparent to the programmer so that he/she need not worry about where the tasks are to be allocated. This, of course, would make parallelism easy to implement with already existing programs as well as new ones.

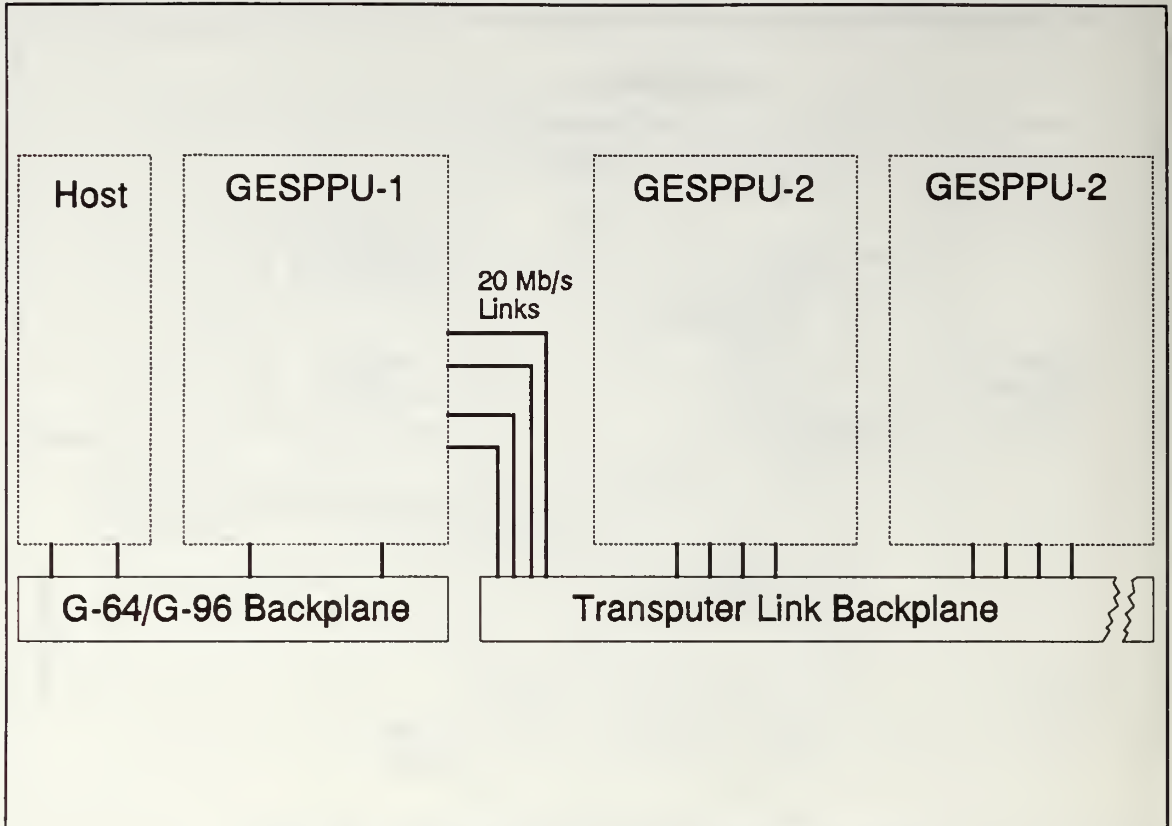


Figure 4 : Transputer network interface with the host

D. ORGANIZATION

In the second chapter, brief descriptions of the tasks used for the AUV-II are given, as well as how they interrelate. This chapter also covers the basics of transputers and transputer networks. Chapter III describes some of the special constructs of Ada that are of interest for this thesis and describes how Ada is interfaced with the transputers. The fourth chapter then describes the construction of the communication layer and the thought behind its desired behavior. The results and performance of the developed layer are presented in Chapter V. Finally, Chapter VI contains conclusions and recommendations for future work and development.

II. PROPOSED AUV-II ON-BOARD MULTIPROCESSOR

A. TASK RELATIONSHIPS FOR THE AUV-II

The AUV-II executes many programs for successful completion of its mission. The inter-relationship of these programs, which from now on will be referred to as tasks, is shown in Figure 5. For the purpose of this thesis, the exact nature of these tasks is not relevant, but the communication scheme is the point of interest. It is important to note that not all the tasks execute at the same rate or in the same order. As previously mentioned, the sonar can be expected to execute at a higher rate and, in addition, there are tasks that execute on an aperiodic basis. Periodic tasks execute at a known frequency of 10 Hz and must finish execution before specific deadlines. Aperiodic tasks on the other hand, execute at random times, as dictated by external events, and provisions must be made to handle the resulting communication from these tasks. For the sake of clarity, a brief description of the all tasks is provided below.

1. Operator

This is an input from the operator prior to the start of a mission. Normally, after the commencement of a mission, the operator no longer has any input.

2. Environmental Database

This is a database maintained in the memory of the vehicle for mapping obstacles that can be, or have been, encountered by the external sensors. It is pre-loaded with known obstacles and then updated by the vehicle's sensor data. Its maintenance is the responsibility of the sonar data processing task. Also, the Plan/Replan Mission task

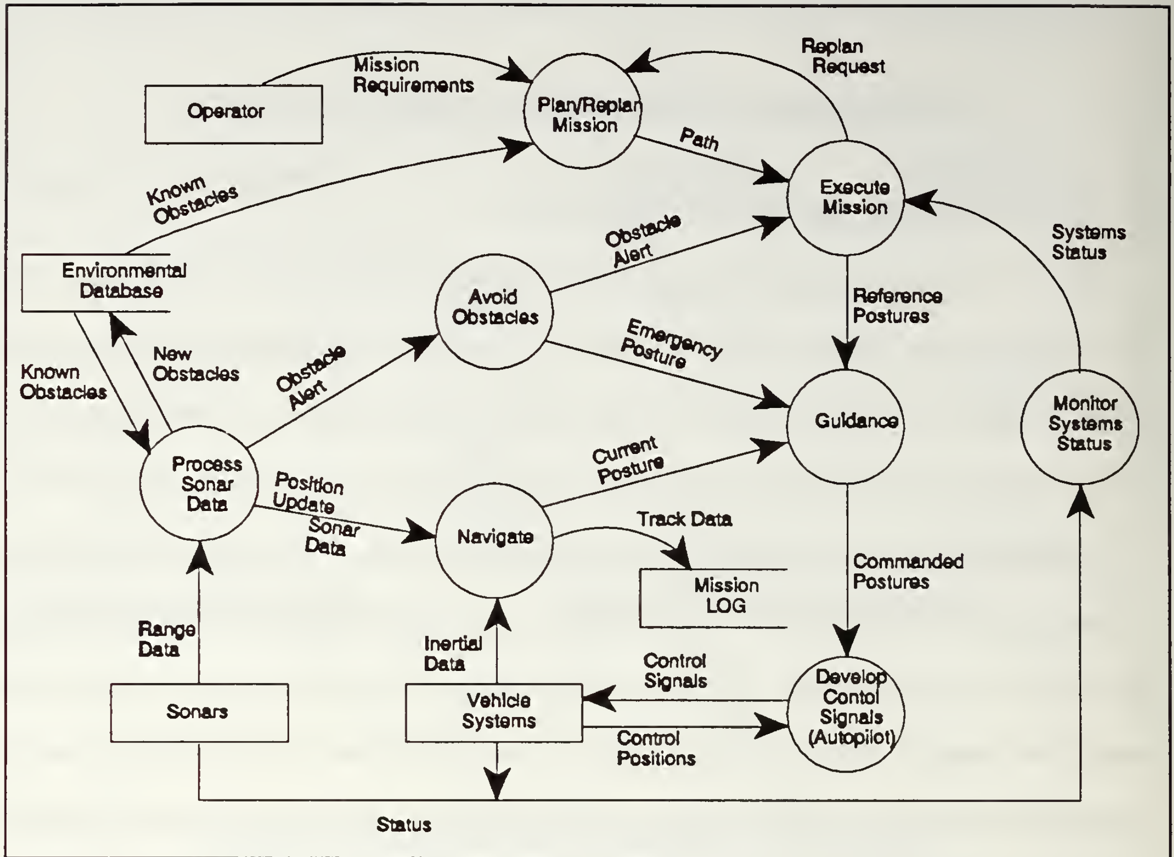


Figure 5 : Data-flow diagram for the AUV [FLOYD 91]

has access to all mapped obstacles for the purposes of creating the best route to the vehicle's destination.

3. Sonars

These are the external sensors that obtain the raw data for processing. The control signals for the sonar are sent via the Vehicle Systems task.

4. Vehicle Systems

This represents all the external systems that control depth, speed, heading, etc. The control signals for these systems come from the Develop Control Signals (Autopilot) task. In turn, the Vehicle Systems returns current status and updates to Autopilot, Sonar, Navigate, and Monitor Systems Status tasks.

5. Mission Log

This is a database for keeping all the desired information of a mission. Post mission reconstruction is achieved through this data.

6. Plan/Replan Mission

This is an aperiodic task that lays out the initial route for the vehicle and generates the waypoints [CLOUTIER 90]. This path is sent to Execute Mission and is then idle until Execute Mission decides that the route needs to be changed, at which time it sends back a request and the mission is replanned.

7. Execute Mission

This process receives the array of waypoints generated by Plan/Replan mission and sends the next single waypoint to Guidance. Execute Mission is also always updated with status reports from Monitor Systems Status and the Avoid Obstacles tasks.

8. Guidance

Guidance receives its instructions from Execute Mission as to the next destination for the vehicle. It is the job of this task to generate the desired values for heading and velocity. These values are then sent to Autopilot. Information from Navigation is also received and compared to the current values in case a correction needs to be made. Finally, input is received from Avoid Obstacles on an aperiodic basis for emergency posture changes.

9. Develop Control Signals (Autopilot)

After receiving the desired heading and speed values from Guidance, the Autopilot generates the necessary control signals for the control surfaces on the vehicle and sends them to Vehicle Systems. In return, Vehicle Systems provides feedback by returning the control positions to Autopilot.

10. **Monitor Systems Status**

This task receives reports from Vehicle Systems on a periodic basis and, from these signals, it determines operating and casualty posture. Its output is sent to Execute Mission.

11. **Navigate**

The Navigate task has the responsibility of determining the current position of the vehicle. It also determines actual heading, velocity, and acceleration [CLOUTIER 90]. It sends its output to Guidance and to Mission Log for recording. The input and inertial data it uses to determine position and vehicle parameters is provided by Sonar and Vehicle Systems.

12. **Process Sonar Data**

The Sonars send raw signals to this task for processing. This is where objects are physically located and mapped. Any previously unknown obstacle is sent to the data base for cataloging. Also, if a possible danger exists, the data is sent to Avoid Obstacle for emergency posture changes if needed. Finally, the data is also sent to Navigate for a position update.

13. **Avoid Obstacle**

This is an aperiodic task that is activated only in the case of a possible emergency. Input from the Process Sonar Data task is received if a possible obstacle is detected. It is the job of this task to generate an emergency posture for the vehicle to assume and send it to Guidance and Execute Mission.

B. TRANSPUTERS

Each of the tasks, outlined in the previous section, has the potential of presenting a complex set of computational requirements. These requirements continue to grow with each new software design as the intelligence of the vehicle grows. To keep the on-board architecture scalable, a transputer based multiprocessor is to be implemented.

1. Overview

The transputer represents a family of microcomputers that have their own local memory and an array of communication links. They operate as a stand alone machine, or as a node in a network interconnected via links [INMOS 89]. When in a network, each transputer operates on its own using only on-chip memory and programs. Communication from one processor to another occurs over the links each of which has a dedicated link interface. The communication interface is implemented in hardware and does not need the processor for its control.

2. Transputer Links

The point to point serial links have several advantages over a common communication bus. Among these is the fact that there is never any contention for use of the line of communication regardless of the number of processors (as system size increases the total bandwidth increases). The second major advantage is that, as the number of processors increases, there is no capacitive load penalty. Finally, regardless of the number of processors, the connection between a subset of processors can be short and local [INMOS REF 87].

The links provide for direct communication between processes on neighboring transputers. Each link consists of two unidirectional signal lines (one going in each direction) and, thus, provides for two communication channels between processors.

Communication across the link uses a link protocol and is accomplished as a sequence of single byte transmissions. This requires only a one byte buffer in the receiving transputer and allows for the same protocol to be used regardless of word size. In each byte, there is a start bit, a stop bit, and a control bit that signifies if the message contains data or an acknowledgement message. If the message contains data, then the control bit is followed by eight data bits [INMOS REF 87].

After each data message is transferred, the next one cannot be sent until an acknowledgement is received from the receiving transputer. Since an acknowledge message can be sent when the start bit of the data message is received, there is no actual delay at the sending end and transmission is continuous.

3. Network Architecture

The transputers to be used for the AUV project (and most other transputers found on the market) have four communication links each. This means that they can be connected in a variety of topologies. One possible network is shown in Figure 6. The network architecture chosen, of course, depends on the implementation as well as the number of nodes in the network. It also, at least partially, determines the complexity of the necessary communication protocol used. Therefore, the communication software consideration should impact the choice of the topology. For this project, the software was developed and tested in the lab on an INMOS B0003 board with four T800 transputers hardwired in a ring is used. A fifth transputer, a T800 with 4 Mbyte external memory, is used as the host for communication with the ring. The configuration used in the lab is nearly identical to a system that incorporates one GESPAC GESPPU-1 and two GESPPU-2 boards with an IBM PC host. For the sake of simplicity, the ring architecture was chosen for this thesis, ignoring the other available links of the transputers.

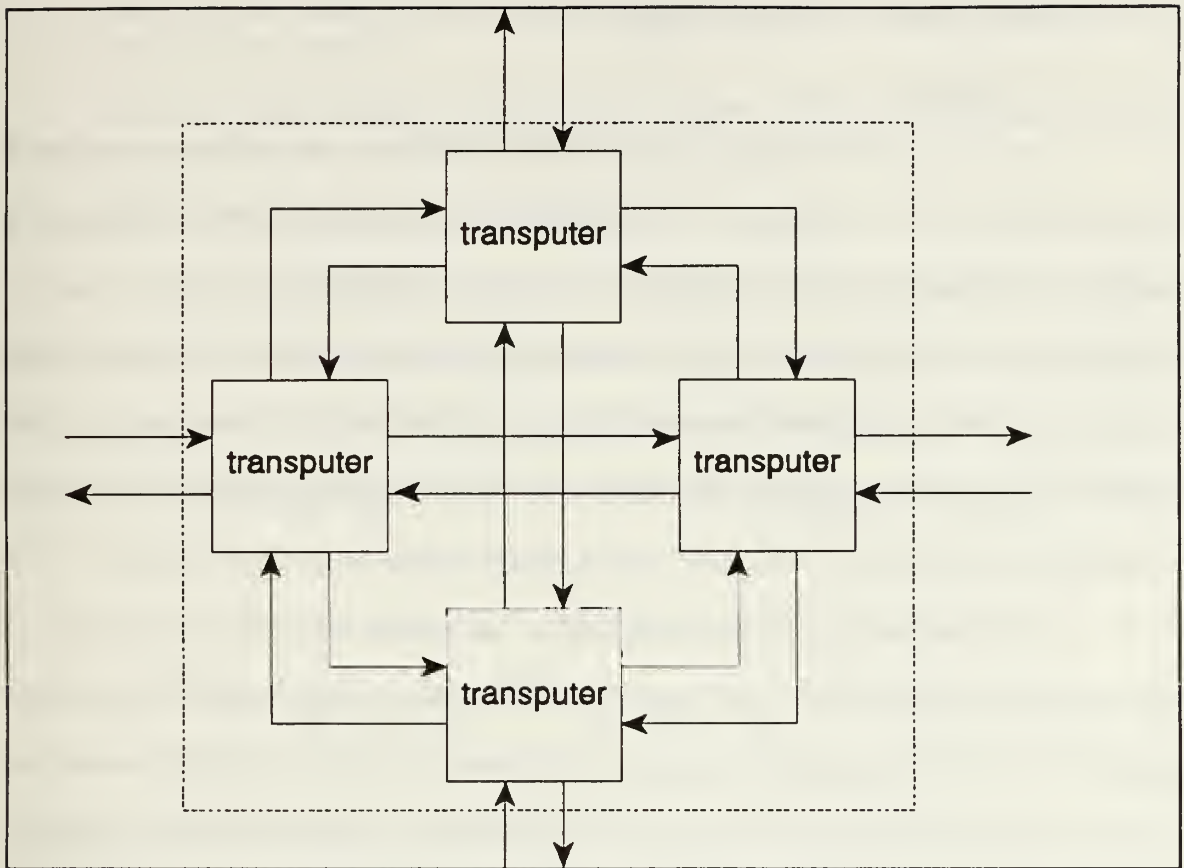


Figure 6 : Four node transputer network

4. Transputer Memory

Transputers are not designed to share memory; instead, each has its own dedicated memory. The transputer is also provided a small amount of on-chip memory for fast access. For the T414, there is 2 Kbytes of static RAM, and, for the T800, this is doubled so that there is 4 Kbytes of static RAM. Also, there is 4 Gbytes of addressable external memory possible [DATABOOK 89].

C. PROGRAMMING TRANSPUTERS

1. OCCAM

The high level language designed for expressing concurrent processes and implementing them on a network of transputers is OCCAM [HOARE 88]. It is rapidly becoming the standard for programming concurrent systems, and in the case of transputers, it is actually executed more or less directly [POUNTAIN 86]. For this reason, OCCAM is often considered the "assembly language of transputers". To execute a program written in any other language, the executable code must be linked together by a framework of channels. This is called a "harness" and is always written in OCCAM.

The harness contains the configuration information to specify the channels and the transputer configuration. It also assigns the separately compiled code to the different processors. When the program is running, it is treated as a single OCCAM process, and as such, can communicate with other OCCAM processes or pseudo-OCCAM processes, such as other programs on separate transputers [ALSYS 90].

2. Interfacing Ada with Transputers

The compiler used for this thesis is the Alsys Ada Compilation System for the Transputer, version 4.4. The process of program development is given in the User's manual as:

1. Create a program library family (groups all related program libraries. There are application families and installation families).

2. Create a program library.

3. Write the source code for the program compilation units.

4. Compile each source unit with the Ada compiler to produce a corresponding object unit.

5. Bind the object units together with the *Run-Time Executive* using the Ada Binder to produce an object module.

6. Link the object module to any required external modules, including the *occam harness*, to produce an executable program.

7. Run and test the program [ALSYS 90].

The interrelationship of all the different files, both user written as well as system generated, is discussed later. The code used for this thesis is given in the Appendix A. Interfacing of Ada with OCCAM is currently a cumbersome task, but many of the above steps can be accomplished in a *Makefile,* and thus, become more or less transparent to the programmer.

III. ADA PROGRAMMING OF A TRANSPUTER NETWORK

A. PRIMITIVES OF ADA AND THEIR USE

One of the principal advantages of Ada is the concurrency constructs that are built into the language. The structure of an Ada program, designed to run in parallel, consists of several tasks that function as programs in their own right and can communicate with other tasks and the main program via entry calls. Inside a program that contains tasks, the separate tasks do not truly run concurrently. Instead, they run in a multitasked mode, since they still reside on a single processor. Processes on a single transputer are time-sliced into approximately 1 ms intervals and it runs its tasks and main program by multitasking [ALSYS 90]. Ada allows for tasks to be given different priorities in which case tasks are executed with highest priority going first. All tasks of the same priority are executed in a round robin fashion [ALSYS 90].

Some of the primitives used in Ada for communication and parallel programming are described below.

1. Entry/Accept Calls

When a task is declared, all the legal entry calls that are accepted by that task are declared along with the specifications of the calls. When another task wishes to send a message to this task, an appropriate entry call must be used. The receiving task does not know the origin of the call, but the sending task must know the destination.

Inside the receiving task, when a point is reached where outside input is desired, an accept statement is used. The task blocks at this point and waits for the proper incoming entry call. The same block occurs at the sending task if the destination task is

not yet ready to receive. When both tasks are ready for communication, a rendezvous occurs and the data is transferred. Both tasks then proceed from that point. If more than one task is attempting to communicate with a single destination then the calls are queued and handled on a first come first served basis.

2. Select Statements

A very powerful construct of Ada is the select statement. This construct allows for alternatives in the execution of the program. The select statement has several uses among which are some that are helpful in building a communications package. The two most commonly used types of this construct are the timed entry calls and the selective waits.

a. Timed Entry Calls

This use of the select statement simply allows for an alternative between an entry call or a delay statement. If the rendezvous occurs before the specified delay, then the entry call is used. Otherwise, if the delay time expires, the task is eligible to continue execution from that point. This type of select statement is the basis of the rotating queue used in this thesis. The most common delay used is zero delay, which means no delay; thus, if the destination is rot ready at the time the call is made, the call is aborted and a different one is tried.

b. Selective Waits

This type of select statement offers two or more alternatives that may or may not have conditions associated with them present for their selection. They are of great use when receiving input to a task when the order of arrival of the different messages is not known or the timing is not known. With this type of select, an input can be received if offered, but if there is none forthcoming at the time, the program can continue without

delay; or if desired, it can wait at this point until an entry call is made that fulfills one of the possible selections.

c. *Select Limitations*

A notable limitation to the select statement, that can be programmed around but only with some difficulty, is that READING and WRITING to the CHANNELS is not a legal alternative. This can lead to a serious problem since the program will stop and wait at any normal read or write statements until they can be processed. This leads to the necessity of devoting additional tasks that were dedicated only to the reading and writing of the channels.

3. Reading and Writing to Channels

In the ALSYS Ada for transputers, there in a generic package called CHANNELS that facilitates the use of the transputer channels. The program treats I/O to a channel as if it were a file. The channels must be declared, and when reading or writing to a channel, the statement must include the channel name. There are no queues of writers for the channels unlike the rendezvous model used in task entry calls; so, when attempting to read or write, the program suspends until the data is available or accepted respectively. This can lead to the serious problem of deadlock.

Deadlock occurs if some or all processes are suspended while waiting for an event that will not occur. A simple example could be if Task A wishes to SEND to Task B, then it will suspend until Task B is ready to receive. But then, if Task B decides it needs to SEND to Task A, it will suspend until Task A is ready to receive. Both tasks are suspended waiting for the other and deadlock has occurred. With any non-trivial communication network, special steps must be taken to avoid deadlock, and to be able to recover the program, should a deadlock situation occur.

Unlike the model used in task entry calls, when using channels, the receiver must know the identity of the channel the data is from; thus, it knows the identity of the sender [ALSYS 90].

4. Delay Statements

This statement is a simple way of delaying a program for any reason or, in the case of the select statement, to declare the amount of time the program will wait for an alternative to occur. Since the actual executable code in the tasks of the AUV-II is not relevant to the testing of the communication software developed here, "dummy" tasks were created that simulated the communication scheme. Delay statements have been used in place of code in the dummy tasks to simulate processing time of the various tasks. They were also used, of course, as mentioned, in the select statements.

5. Read_Or_Fail / Write_Or_Fail Statements

These two statements are constructs that partially compensate for the deficiency mentioned above under the select statements. With these, the programmer can designate a particular time, as read from the on-chip clock, that the process will wait for until it declares the read or write attempt a failure. After the statement is executed, a variable can be checked to find out if the attempt was successful or not. These statements can be used to insure that deadlock cannot occur, but increases the size of the code necessary since an entire new library is needed to implement them.

B. ADA AND ITS USE WITH TRANSPUTERS

From the start, one of the goals of this thesis was to use Ada on transputers. The first reason for this goal is that Ada has been specifically designed as a concurrent language, making it a logical choice for use on concurrent processors. Another important

point is that Ada, is the currently the adopted DoD standard and all software development for the DoD is encouraged to use it. Finally, Ada has the communication primitives necessary to utilize the transputer links efficiently. The use of any pragmas to other languages is not required.

1. Design Considerations for Ada

Each transputer is required to have an Ada program that can be compiled and run as a "stand alone" program. This means that the tasks in Ada actually only run in a multitasked fashion since all the tasks in the program must run on the same processor. On the transputers, this is handled normally by time-slicing the processor into approximately 1ms intervals. This can be modified with a priority system inside the program if desired. The real concurrency is the actual programs that run in parallel on the different transputers.

2. Communications Primitives Available in Ada

For the Alsys system used for this thesis, there is a package called CHANNELS that provides the necessary routines for the channels declared in the OCCAM harnesses to be used inside the Ada program. The procedure calls are simple READ and WRITE statements with the designated channel (declared in the program) as one of the arguments. These statements may be used anywhere in the program any number of times.

A problem arises when there is unrestrained use of these statements, resulting in deadlocking the program. The compiler does not notify the programmer if, due to communications on a channel not being synchronized, a deadlock will occur and no error message is generated if it does occur during run-time. This limitation leaves the problem of insuring that all communications take place in such a manner that the deadlock situation

where a sender is waiting to send but the wrong receiver is waiting to receive, never arises, as the responsibility of the programmer..

The basic task communication is accomplished through a rendezvous and multiple task calls are queued [ALSYS 90]. If the protocol used in the task calls could be simulated and used in the program communications, then the problem of deadlock could easily be solved.

3. Ada as a DoD Standard

Ada was created as the result of the United Sates Department of Defense's attempt to standardize software used in the DoD. In the late 70's, and the early 80's, Ada was accepted as the standard in the USA, and in 1987, it was accepted as an international ISO standard [SKANSHOLM 89]. Ada is still currently the DoD standard. This, in addition to the already mentioned advantages, made it an ideal choice for this project.

C. INTERFACING ADA TO TRANSPUTERS USING OCCAM

The process of binding, linking, and loading a system of Ada programs for a transputer network is currently rather arduous. OCCAM is a programming language that has the capability of total concurrency and has a special relationship with the transputer. As mentioned earlier, transputers can simply be thought of as the hardware implementation of OCCAM [TRANS 89]. As a result, all higher level languages used on transputers are invoked through OCCAM *harnesses*.

1. The OCCAM Harness

There are a variety of programs that are provided for the Alsys compilation system in the *Occam2 Toolset* which is a set of tools written by INMOS [ALSYS SYS 90].

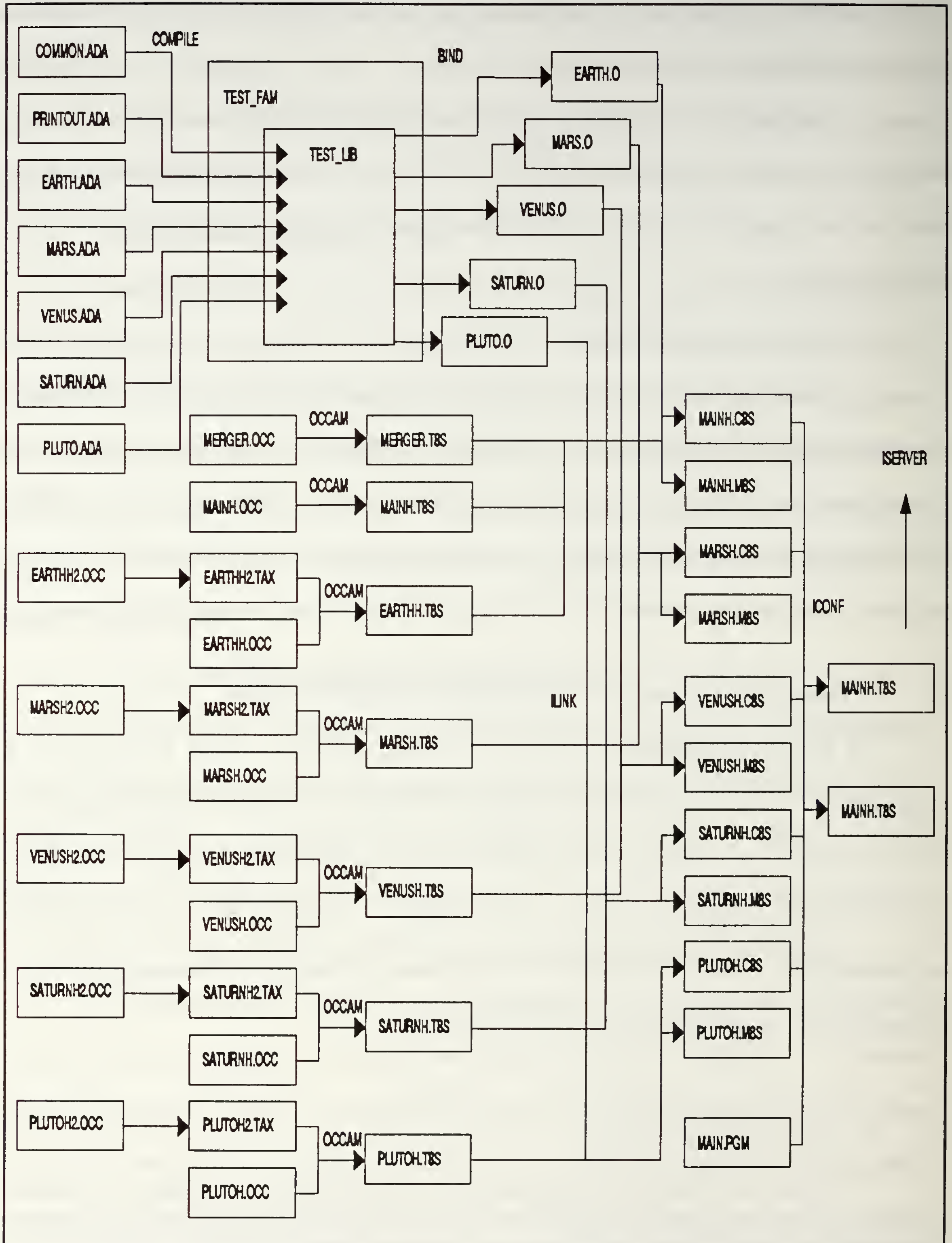


Figure 7 : Relationship between files for Ada on transputers

Table I : Description by file extension

Extension		Contents	User Written
*.ADA	--	Ada source code (text)	YES
*.OCC	--	OCCAM source code (text)	YES
*.PGM	--	OCCAM source hardware configuration (text)	YES
*.O	--	Compiled Ada	NO
*.TAX	--	OCCAM compiled in T800 (TA) UNIVERSAL mode (X)	NO
*.DSC	--	Configured code descriptor file (text)	NO
*.BTL	--	OCCAM module, bootable by iserver	NO
*.x8S	--	OCCAM compiled in T800 STOP mode	NO
*.Cxx	--	Linked OCCAM code (not bootable)	NO
*.Mxx	--	OCCAM configuration map (text)	NO
*.Txx	--	Compiled OCCAM	NO

Since the actual mechanics of binding, linking, and loading are not of interest for the purpose of this paper, only the interrelationship of all the files, both written and generated, is provided in Figure 7. Table 1 provides a key to the different file extensions and also shows which of the files must be written by the programmer and which are generated.

What is important to know is that the assignment of the hardware channels to the transputer links, as well as the processors to the programs, occurs in the harness. This also applies to the assignments of the allowed memory to use for the work space. A copy of all the harnesses and text files used for this thesis are provided in the Appendices.

2. Static Allocation

An important as well as limiting factor is the fact that the OCCAM source hardware configuration given in the *.PGM file of the harness is static. The consequences of this are that it is not currently possible to write a program that will allocate tasks to an

appropriate processor (i.e. this must be done by hand). Since this is implementation dependent, it may be a future improvement needed in the system.

IV. BUILDING A COMMUNICATIONS PACKAGE

A. OBJECTIVES AND DESIRED BEHAVIOR

The long term goals of this project are to make it possible for a programmer to write a single task oriented program, and run it on a network of transputers without knowing

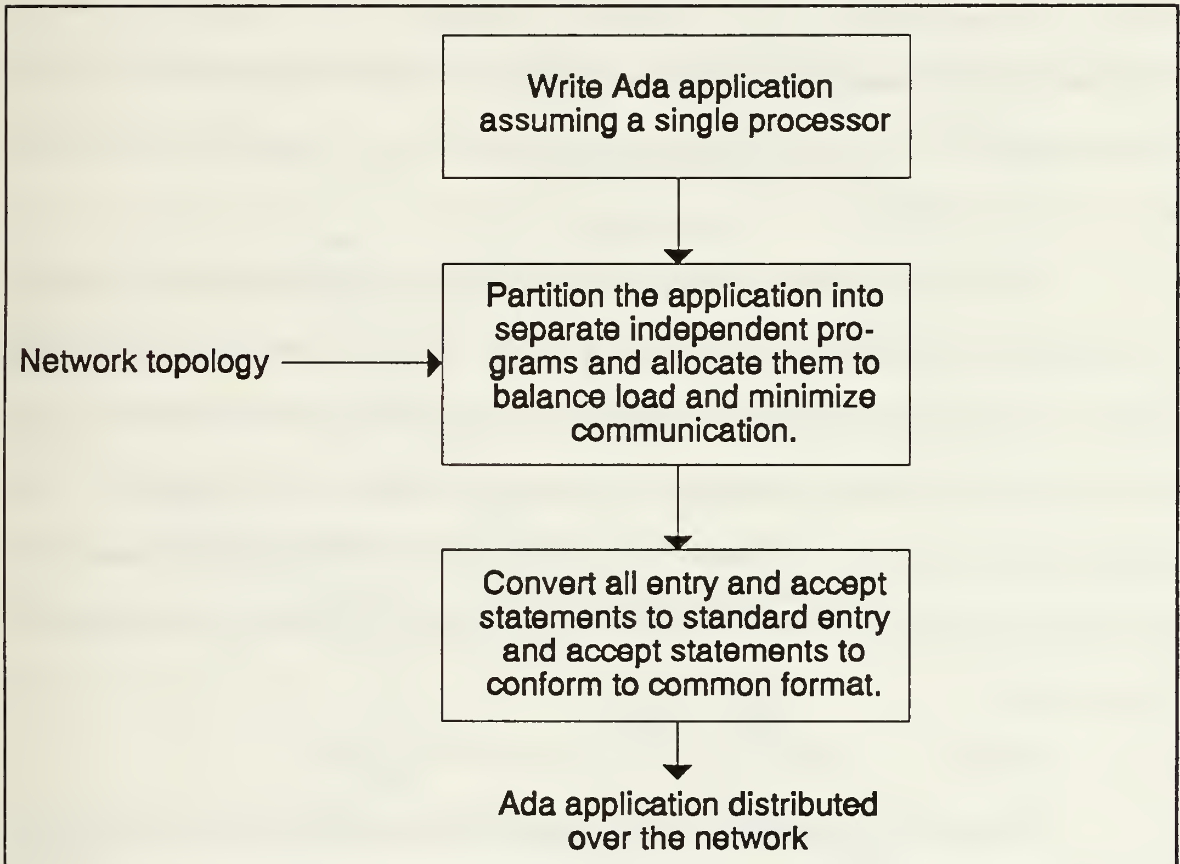


Figure 8 : Overall functionality of software layer

anything about the network or its communications. As mentioned earlier, hiding the network from the programmer in this manner requires an intermediate software layer. The

conceptualized application development procedure using this software layer is shown in Figure 8. The first step in this is creating a communication scheme for a network that will enable the intertask communication to be location invariant since the programmer will not know the final locations of the tasks when writing the program.

1. Design Criteria

Important factors that went into the design of this layer included reliability, deadlock avoidance, and speed of message delivery. In any network communication, reliability is a major criterion. For the purpose of this thesis, the level of reliability desired is directed towards guaranteeing that a message will be delivered even at the cost of extended time delays. No provisions are made to recover from messages lost or damaged due to hardware failures.

The structure of the software communications layer is designed in an attempt to minimize the possibility of a deadlock situation. Even so, timers are used in the program to insure that the processes responsible for communication will not hang up at any one event, thus insuring a recovery from a deadlock situation should one occur. This design to prevent deadlock aids speed of message delivery also since communication lines are not allowed to stay dormant for extended time periods when there are messages that require delivery.

2. Goals

The goals of the program and its behavior are closely related to the design criteria but are more specific. They are:

- Intertask communication is location invariant (reliability or operability not are not affected by where the task is located).

- Communication between local and non-local tasks uses the same syntax; this is required to attain the previous goal.

- Messages are delivered in a timely and reliable fashion.

- Ada semantics of blocking rendezvous mechanism is preserved.

B. SOFTWARE COMMUNICATION LAYER

1. Concept

The block diagram in Figure 9 shows the structure of the software tasks used

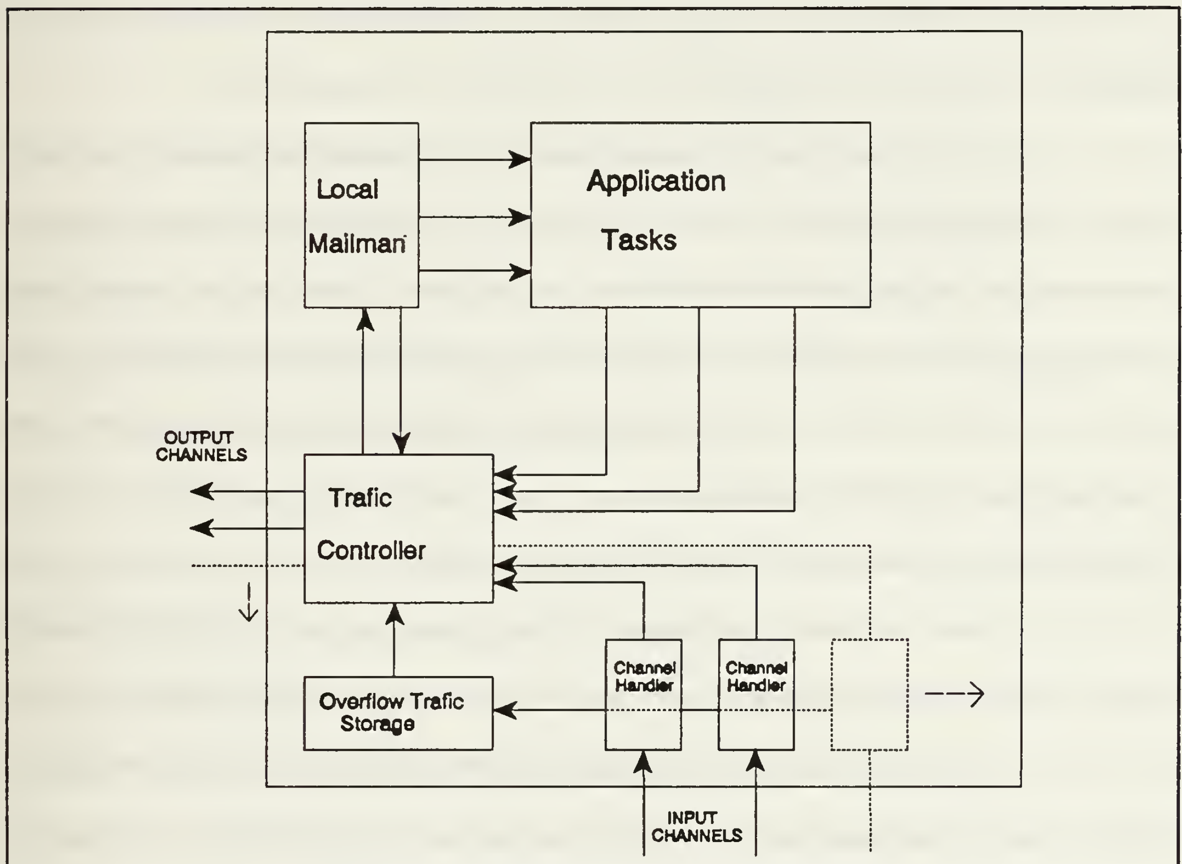


Figure 9 : Communication layer structure

to create the communications package. Ideally, this structure could handle any number

of input and output channels. Since each channel handler represents a dedicated task for the transputer, the task load on a transputer could be prohibitive for transputers with more than the normal four links. A description of the various blocks is provided below.

a. Channel Handler

This is the simplest of the communication tasks. Its sole function is to read data from a single channel. When a Channel Handler receives a message it simply passes it to Traffic Controller. If Traffic Controller is busy, in order to keep the incoming channel open for further messages, the message is placed in Overflow Traffic Storage. Once a Channel Handler hands off its message, it returns to a wait state for more data from the channel.

The reason this function is required to be accomplished in a dedicated task is that the select statement in Ada does not allow a READ statement to be an alternative. This means that the program cannot check a channel to see if there is data there without actually doing a READ. If a READ is executed, the program will go into an indefinite wait period until data is available, thus precluding the execution of any statements further on in the program. By putting the Read statement in a task by itself, the frequency of input does not control the frequency of any other task execution.

As mentioned in Chapter III, Ada does provide a READ_OR_FAIL statement that enables the programmer to set a limit to which the READ will be attempted, after which it will be aborted. The inefficiencies and the additional algorithms needed to use this statement often precluded its use. It was decided that the additional task load is a better alternative in the case of reading from a channel.

b. Traffic Controller

The purpose of this task is to direct all messages in the proper direction. It is here that the location of the tasks must be known. For the communication architecture used for this thesis, the only determination to be made is if the destination is local or not. In other architectures, if the destination is not local then Traffic Controller may need to decide to which output channel to send the message. Since a ring communication scheme is used, there is only one output channel for each transputer allowed. The one exception to this is in the program MARS in which output is also sent to the program EARTH for I/O to the screen. These programs are described later.

If messages are determined to be destined to local tasks, then Traffic Controller directs the message to Local Mailman. If Local Mailman is full, then the message is automatically sent back to the loop with the intention of catching it the next time around. Again, this is done so that Traffic Controller will never be stuck with a message. The drawback to this is that, in heavy traffic, it is possible that a message passes around the loop several times before being delivered.

Messages are received by Traffic Controller from all local sources including the application tasks, Local Mailman, all Channel Handlers, and Overflow Traffic Storage. Therefore, for this concept to be successful, it is critical that the channels remain open (i.e. a message cannot get stuck on a channel, and thus, prevent the Traffic Controller from writing to it) so that Traffic Controller can process messages without delay.

As mentioned before, the task will wait at the write statement until it is able to execute it regardless of the delay involved. This is why Channel Handlers will not hold a message for any amount of time, but rather, attempt to send the message to Traffic Controller. If that fails, the message is immediately put in the Overflow Traffic Storage.

In this way, we are assured that Traffic Controller will incur no time delay when attempting to write to a channel and a deadlock situation will be avoided.

A further safeguard for keeping the channels free is that Traffic Controller will always prefer to accept input from Channel Handlers. If, and only if, no other input is offered from these tasks will input be accepted from the other local routines.

c. *Overflow Traffic Storage*

This is a simple task that stores messages and sends them to Traffic Controller on a first-in-first-out basis. The storage facility in this task is a static array, and therefore, it has a maximum size. At this point in the development, the maximum size needed can only be determined by the application. During the testing performed for this thesis, it was found that this task was rarely even used; therefore, a very small maximum size was found to be sufficient.

An item of interest that is a result of the priority system of Traffic Handler is that a message coming in from a channel into a Channel Handler has a higher priority to be accepted into Traffic Handler. If it fails to be accepted, then it is passed to Overflow Traffic Storage and its priority is then lowered. This means that it is possible that two messages passing through a transputer may actually be reordered if the first of the two were placed in overflow and second passed on through unhindered. The possible reordering of messages was not seen to be a problem since the beginning order was random in the first place.

d. *Local Mailman*

The purpose of this task is to provide a rotating queue that sends messages to their final destinations when they are ready to be received. The task receives a message and stores it in a static array. It then rotates through the array looking in each slot to see

if it has a message to send. If the slot is full, then a quick attempt is made to send it. If the receiving task is not ready, then the attempt fails and Local Mailman skips the slot and looks for another. If the receiving task is ready to accept, then the message is sent, the slot is emptied, and an acknowledgment message is sent back to the sending task via Traffic Controller.

As is the case with Overflow Traffic Storage, the array for the queue is static; so again, there is a maximum number of messages that can be stored. In contrast to Overflow Traffic Storage, the maximum size chosen did have a measurable affect on the average message delivery time. This will be discussed in the next chapter.

Local mailman has access to all the necessary entry calls for the application tasks on its transputer. For this thesis, this was handled in the form of a separate procedure called by Local Mailman which is tailored for individual transputers. In later developments, this procedure will be generated by a higher level program.

e. Application Tasks

These are all the tasks written by the programmer for the application program. All tasks receive messages from Local Mailman and send outgoing messages to Traffic Controller (even if the destination task is local). This leads to the necessity of standardizing the message format used by all tasks.

2. Message Format

The message format used could vary from application to application with only a few common requirements. The items that must always be present in the message are the destination task, the sending task, and the entry call to be used. The reason for the destination task and the entry call to be included is obvious. The reason for the sending task to be included is that Local Mailman of the receiving transputer should know who to

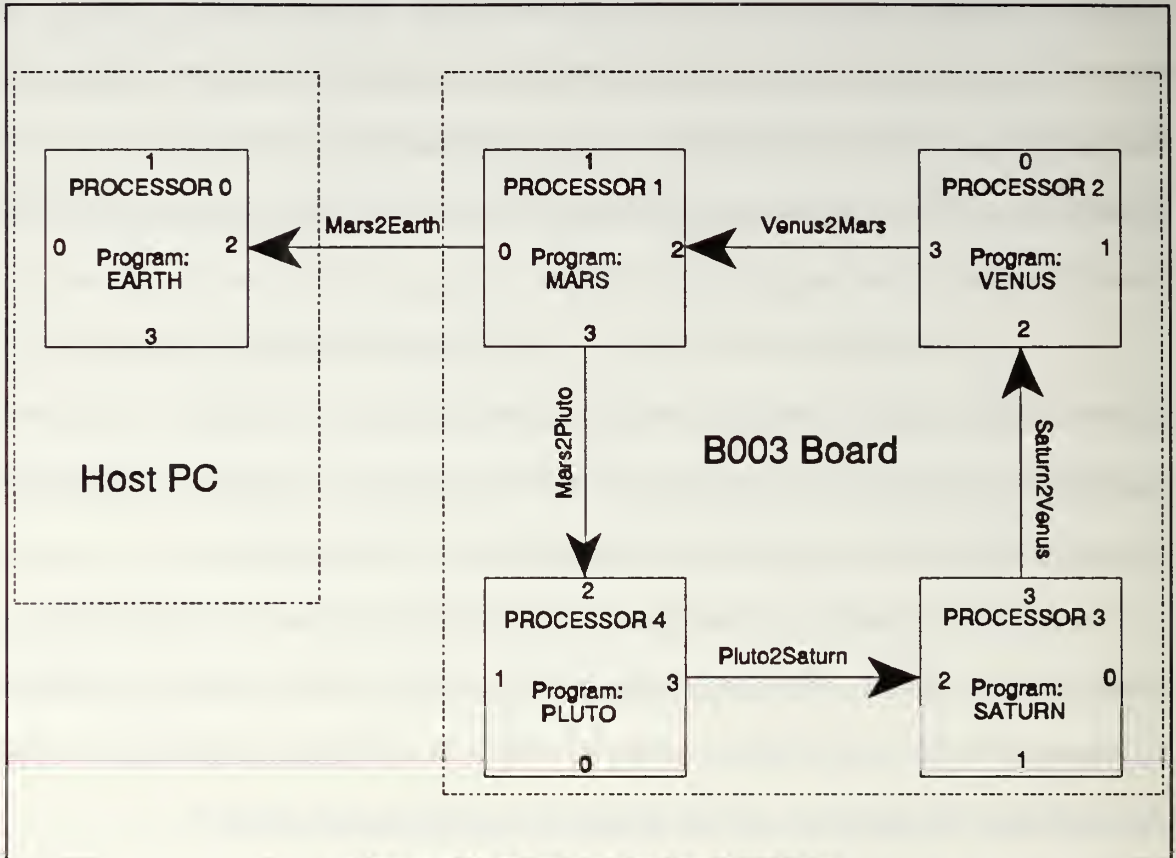


Figure 10 : Communication topology

send the returning acknowledgement message. As described later in this chapter, this acknowledgement message is required to capture the Ada rendezvous semantics. The remainder of the message is application dependent.

For this thesis, the message was in the form of a record with entries mentioned above along with a time entry for timing message delivery times, as well as two arrays for data (these arrays were initialized but not actually used for any purpose).

3. Communication Architecture

The topology used for communication for this thesis was a simple ring. It was chosen for its simplicity and the intention that, if the software layer can be made to work

with a ring, then in later developments, more complex topologies can be easily incorporated. Figure 10 shows the topology and the programs resident on the different transputers.

C. CAPTURING THE RENDEZVOUS SEMANTICS

In Ada, communication between tasks within a program is accomplished through accept and entry calls. A task wishing to receive a message has an accept statement; and when such a statement is reached, the task blocks until a message with the proper entry call is received (regardless of sender identity). A task wishing to send a message names the destination task and the entry call with the message as the argument. If the receiving task is not yet ready to accept the message, the sender blocks until the message can be sent. The sane holds when the receiver reaches the accept statement first. When both communicating tasks are at their respective statements, a rendezvous occurs. At this point, the message is transferred, and both tasks continue their execution from that point.

For the communication software developed here, it is desired to preserve this mechanism. Since, in the case of location invariant communication, the communicating tasks cannot be assumed to be co-located on a processor, an actual rendezvous as discussed above cannot occur. To simulate it, an acknowledgment message is used.

When a message is known to have reached its destination, an acknowledgment message is generated and sent to the originating task. The originating task, as part of the communication protocol, has an accept statement immediately following any entry call made to another task. This prevents the sending task from continuing in its processing until it is assured that its message has been received. This captures the rendezvous mechanism completely except for the fact that the receiving task will commence its

execution slightly before the sender since some amount of time is required for the delivery of the acknowledgement message.

The acknowledgment message in the software developed for this thesis is generated in the task Local Mailman since it is there that it is first known if a message has been successfully delivered. It is generated using a generic message with the originating task used as the destination, and an entry call common to all tasks for receiving acknowledgments. Local Mailman will not send an acknowledgment if the message delivered, was itself an acknowledgment.

V. PERFORMANCE ANALYSIS

A. SIMULATION ARCHITECTURE

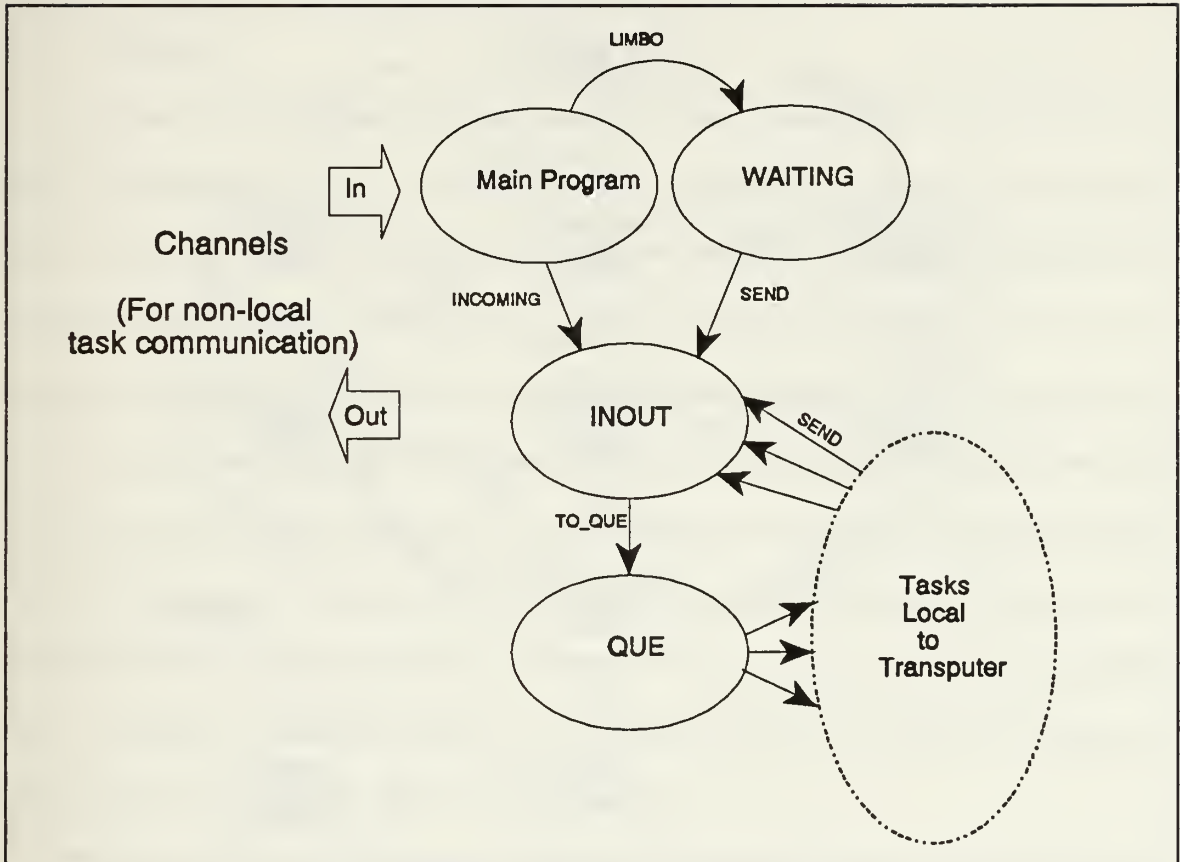


Figure 11 : Message flow at a transputer

To test the tasks written for the communication software layer, a set of tasks similar to the one seen in the AUV-II data-flow diagram is used. Figure 11 shows the interrelationships of the tasks that handle the communications and are common to all transputers and Figure 12 shows the task interrelationships as set up for the testing of the communications. The entry calls are shown between the two tasks and the transputer that the task is running on is shown below the task name.

The components of Figure 11 relate to the software layer structure previously shown in Figure 9 as follows: the Main Program shown serves as the Channel Handler with the task WAITING serving as the Overflow Traffic Storage; the function of Traffic Controller is handled by the task INOUT; and Local Mailman is QUE.

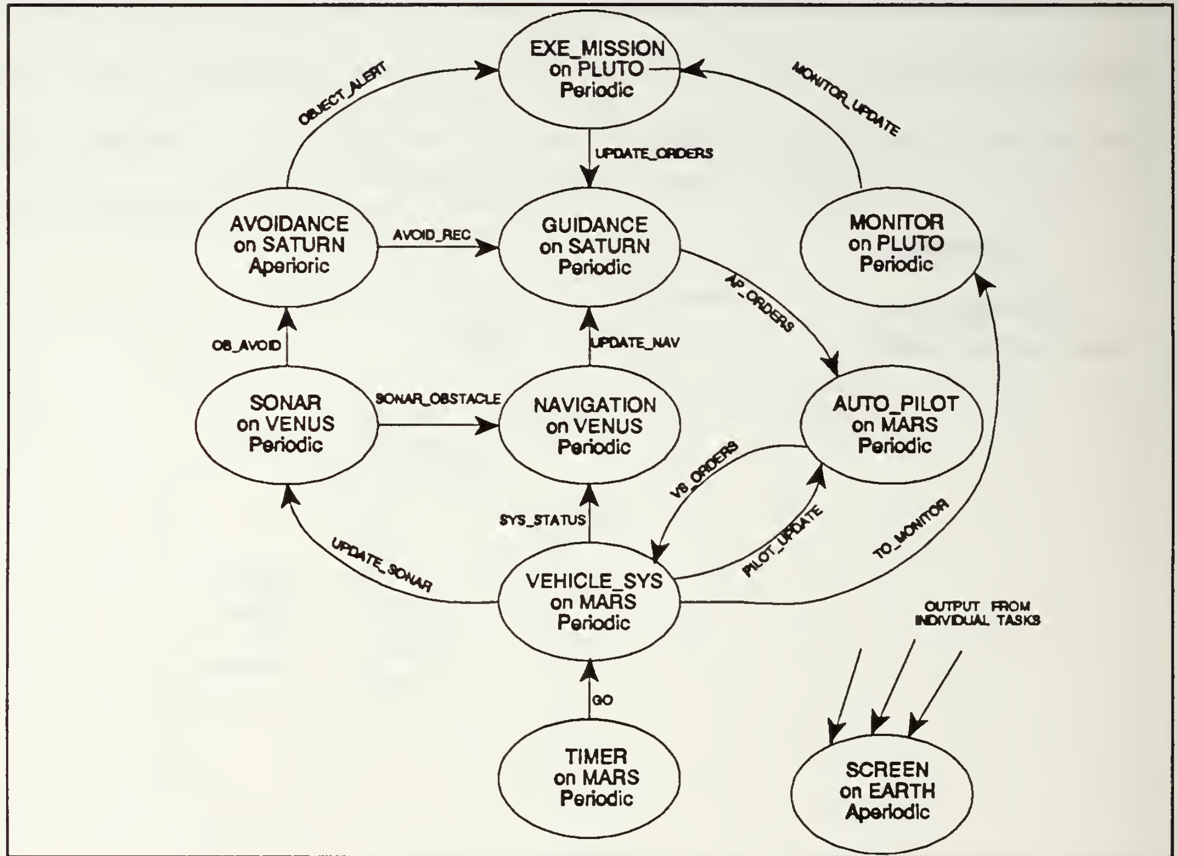


Figure 12 : Simulation data-flow

A task added to control the frequency of execution is also shown in the Figure 12 called TIMER. This task simply outputs a GO to VEHICLE_SYS at a predesignated rate and the execution of the latter task cannot proceed until it is received. The main program in Ada functions much as does one of the program's tasks, so it can be used for one of the communication layer tasks. Here it is used for the Channel Handler since for this topology only one is needed. Finally, the task SCREEN in the earth program is simply an I/O

routine that will decode a message and print it out regardless of the origin. This was found to be the best way to troubleshoot the programs and output performance measurements since only the host transputer has the capability to output information to the screen.

B. RESULTS

In testing the software developed for this thesis, a series of runs were made, each with one hundred iterations. A single iteration consists of all eight AUV-II simulation tasks executing once, and all thirteen messages involved sent and acknowledged (the task TIMER is not included in this, although it controls the frequency). The number of one hundred was arbitrarily chosen and was considered to be high enough to approximate continuous program execution. The first significant result is that deadlock did not occur for any number of program iterations. This was tested by varying the maximum queue size and running the program for one hundred iterations at maximum frequency. Even when the queue size was dropped to a maximum of only two, the one hundred iterations ran to completion without deadlock. This showed that even when the queue was overloaded, messages were still reliably delivered.

The second major achievement is that the location invariant communication was achieved. Table 2 shows the original location and the final locations of the tasks in relation to the programs on the transputers. The final locations are considered to be more optimal than the beginning positions because as many communicating tasks as possible are collocated. In both cases the communication was conducted without deadlock or lost messages. It is important to note that the average time per iteration was significantly higher for the first task placement.

Table II : Task locations

Task	Beginning Location	End Location
AUTO_PILOT	MARS	MARS
AVOIDANCE	PLUTO	SATURN
EXE_MISSION	SATURN	PLUTO
GUIDANCE	VENUS	SATURN
MONITOR	PLUTO	PLUTO
NAVIGATION	SATURN	VENUS
SONAR	MARS	VENUS
VEHICLE_SYS	VENUS	MARS

The reason that the second placement is more advantageous than the first is that tasks which communicate with each other were placed on the same transputer. In the first placement, the two tasks on each transputer never communicated with each other, only with the tasks on the other transputers. In the ring topology, it turns out that whether the destination transputer is one hop away or three, the total number of hops needed for communication is four. This is due to the fact that all communications need the acknowledgement to be completed, so one full trip around the ring is required to deliver both the message and the acknowledgement. Therefore, the time of message delivery is made faster is only if the destination is on the same transputer as that of the point of origin.

The final results are the different iteration times for one full run, in which each task is executed once and all communications occur once. Although these times were not what was hoped for, they do provide some interesting results. Table 3 shows the average times measured for a single iteration (averaged over one hundred iterations) with the two different task allocations. For this measurement, a queue size of fifteen was used. The decrease in time per iteration shows expected results. Table 4 shows the times measured

for the optimal task allocation with varying queue size. As can be easily seen, the optimal queue size for this particular communication scheme is quite small. This is due to the fact that, at any one time, there are actually only a few messages en route. When the queue size is too large, then there is time wasted in checking the empty slots looking for messages to deliver. If the queue is too small, then time is wasted when a message is sent around the loop again due to the queue being full. The minimum queue size that the software is designed to handle is two.

Table III : Average iteration times versus task allocations

First Allocation	332 ms
Second Allocation	235 ms

Table 5 shows the timing results when the time between loop iterations is controlled by the task TIMER. The times shown in the table represent the average time for the completion of a complete iteration. What is of interest is that when the delay between iterations becomes less than the execution time (thus, making the TIMER task the controlling factor instead of the iteration completion), then the reported time is 17.5 ms. This time does not represent a full iteration, but instead, represents the time for the task VEHICLE_SYS to send and receive the acknowledgements for four messages (three of which are not local). This represents one third of the communications necessary for a single iteration. Intuitively, this would seem to mean that one iteration should be able to occur in about 55 ms or less; this brings into question why, instead, it takes over two hundred ms.

Table IV : Average iteration time versus queue size

Queue Size	Time (ms)
25	253
20	239
15	235
10	229
5	224
2	225

Table V : Measured iteration times when TIMER frequency is controlled

Delay between Iterations (ms)	Time (ms)
0	224
100	223
200	219
250	17.5
300	17.5

C. LIMITS ON PERFORMANCE

The undesirable aspects of the software layer handling the communications between tasks is that it takes up a lot of memory and time. As an example, for the ring architecture used for this thesis, let us examine the number of tasks needed for one message to be delivered to a non-local destination. It is handled by a Traffic Controller six times (this includes the acknowledgement message), a Channel Handler four times, and a Local Mailman twice. This adds twelve to the number of tasks that handle a single message for delivery to a single destination. If direct routing were used, this could be reduced to eight tasks. Reducing this number could be one place for further optimization.

Taking these calculations a step further and comparing them to the measured results, the difference direct routing could make becomes apparent. In a single iteration, there are

thirteen messages sent and acknowledged. This means that, in case of the sub-optimal allocation above, there were at least 156 tasks that were executed for message delivery and at least 52 links were crossed. In the optimal allocation, there were 116 tasks executed and 32 links crossed with a resulting improvement of about 100 ms. If direct routing were used, only 84 tasks would be executed, in the optimal case, and 16 links would be crossed.

It was found that the task load on the transputers, when reduced, had only a slight effect on the iteration time. Also, the length of the formatted message even when reduced by over fifty percent, had no measurable affect on iteration times. Lastly, a reduction in message traffic was accomplished by reducing the frequency of reporting the iteration times. This also had negligible effect on the average delivery time.

Our results indicate that the communication layer only supports an execution rate of about four hertz. However, this does not indicate the potential of the approach as being limited. In the simulation tasks, no output is allowed to be sent until the proper inputs were received. This means that only one iteration can be in progress at a time, and the concurrent nature of the transputers is not being used to its fullest extent. If the data-flow were pipelined, it will be possible to reach the required frequency easily.

VI. CONCLUSIONS AND FUTURE WORK

A. CONCLUSIONS

The results of this work show that intertask communications can be reliably handled by a software layer. The communication can be made to be location invariant with the software architecture presented. Due to the large number of tasks needed to implement the communication, there appears to be an excess amount of time spent in message delivery. Therefore, any modifications that can reduce the number of tasks that handle a message is desirable. The most obvious way to reduce this number is by employing direct routing. However, the real limitation is in the intertask communication which allows only one iteration to be in progress at a time.

Finally, the results show that although the communication may be location invariant, the average time for message delivery is not. This means that finding the optimal placement of tasks on the transputer network is important.

B. FUTURE WORK

1. Higher Level Program

The next step in the development of the concurrent programming package is to create a program that will allocate the tasks to transputers to optimize a suitable criterion and then write the mapping routines for the individual transputers. The generation of the task calling procedure, used in Local Mailman, on each transputer would also be the responsibility of this program. This program should also be able to decide upon the common message format and then change all accept and entry statements to

conform to the standard. This could be accomplished by a filter that changes the entry/accept statements to the appropriate Ada procedure calls that use the message routing supported by the communications layer.

The items in this thesis that will have to be rewritten or handled by such a program are as follows:

- The procedure SEND_IT must be generated (This is the procedure that contains all the entry calls to the local tasks).
- The procedure IS_IT_HERE must be rewritten (This is the procedure that is in the common package and determines if a task is local on not. If a different topology is used, an equivalent algorithm to determine path of message propagation must be written).
- All tasks must be allocated to a set of independent programs.
- The record MESSAGE_FORM must be written to accommodate all requirements for task communication on the network.
- The task SCREEN must be written to handle all desired I/O to the screen.
- All entry and accept statements must be made compatible with the record MESSAGE_FORM.

2. Difficulties

One of the conclusions from this thesis is that Ada may not be the best environment to implement a task communication layer. Ada was not designed to act as a vehicle to implement an operating system; but in this software layer, there are many operating system like functions programmed using it. Also, the lack of dynamic memory allocation in Ada makes the writing of a queue cumbersome and inefficient. The other major drawback of Ada encountered in this thesis was the fact that using a READ or WRITE statement was not a legal alternative in a SELECT statement. These factors lead to a substantial message passing overhead.

The final drawback encountered was the compiler itself. Compilation of the programs is excessively time consuming. The creation of joint invoke files would reduce total time of compilation when the programs for several transputers are to be compiled. As the number of transputers in a network rises, this will become more and more desirable.

3. Parallelism

Finally, in order to take advantage of a network of processors, there must be more than one possible concurrent sequence of events. The smaller the interaction between concurrent processes, the higher will be the speed up and the greater will be the use of the capabilities of a multiprocessor. This will be accomplished only through careful programming practices and experimentation that exploit parallelism.

The software architecture presented successfully accomplished the goal of making tasks on a network location invariant which is vital in making parallel programming easy to implement. Another mandatory requirement met was the successful avoidance of the severe problem of deadlock.

APPENDIX A: OCCAM SOURCE CODE

A. OCCAM HARNESS FILES

These files are the OCCAM source files for the harness used on the transputers. They are all quite similar with the exception of the main harness used on the root transputer which also incorporates the needed system communications. These harnesses were derived from the examples given in the Alsys Ada User Manuals.

1. Main Harness

For the main harness, unlike the other harnesses, the are two occam files combined to make the harness. These two files are the EARTHH.OCC (which represents the normal one found on the other transputers) and MERGER.OCC which enables the host functions. The combination of these files make up the main harness.

a. Main

```
-- File: mainh.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

-- These are the declarations used for the occam channels. ToFiler
-- and FromFiler are the channels used for the system control
-- functions.

PROC main.harness (CHAN OF SP  FromFiler, ToFiler,
                  CHAN OF INT Mars2Earth, Earth2Mars,
                  []INT FreeMemory)

  #USE "hostio.lib"
  #USE "earthh.t8s"
  #USE "merger.t8s"

  [1]CHAN OF ANY Debug:
```

```
[2]CHAN OF SP FromAda, ToAda:
CHAN OF BOOL StopDebug, StopMultiplexor:
SEQ

  PAR

    -- A multiplexor to combine the debug and normal output.
    so.multiplexor (FromFiler, ToFiler, FromAda, ToAda,    StopMultiplexor)

    -- A debug channel merger.
    debug.merger (ToAda[0], FromAda[0], Debug, StopDebug)

    -- A process to invoke the earth program.
    ws IS FreeMemory:
    SEQ
      earth.harness (FromAda[1], ToAda[1], Debug[0], Mars2Earth,    Earth2Mars, ws)
      StopDebug ! FALSE
      StopMultiplexor ! FALSE

  so.exit (FromFiler, ToFiler, sps.success)
:
```

b. Merger

The following file is part of the main harness and is used for multiplexing the control functions over a single channel. It was taken directly from the Alsys Ada User Manual and included without change in the programs for this thesis.

```
-- File: merger.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC debug.merger (CHAN OF SP FromFiler, ToFiler,
                   []CHAN OF ANY Debug,
                   CHAN OF BOOL Stop)

  #USE "hostio.lib"

  -- A debug channel merger and blocker.

  VAL max.debug IS 20:
  VAL number.of.debug IS SIZE Debug:
```

```
INT line.index:
[256]BYTE line.buffer:
BYTE value, r:
BOOL running, reset, s:
[max.debug]BOOL mask:
VAL BYTE line.feed IS 10 (BYTE):
SEQ
  SEQ i = 0 FOR number.of.debug
    mask[i] := TRUE
  running := TRUE
  reset := FALSE
  line.index := 0
  WHILE running
    PRI ALT
      ALT i = 0 FOR number.of.debug
        mask[i] & Debug[i] ? value
          SEQ
            IF
              value = line.feed
                SEQ
                  -- Send the complete line.
                  so.puts (FromFiler, ToFiler, spid.stdout,
                    [line.buffer FROM 0 FOR line.index], r)
                  line.index := 0
                  mask [i] := FALSE
                  reset := TRUE
              TRUE
                SEQ
                  -- Add character to line.
                  line.buffer[line.index] := value
                  line.index := line.index + 1
      reset & SKIP
        SEQ
          reset := FALSE
          SEQ i = 0 FOR number.of.debug
            mask[i] := TRUE
      Stop ? s
        running := FALSE
:
```

c. Earth

This OCCAM source file is nearly identical to that found on the remaining transputers. It is here that the OCCAM channels are specified and loaded into the program.

```
-- File: earthh.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC earth.harness (CHAN OF SP  FromAda, ToAda,
                    CHAN OF ANY Debug,
                    CHAN OF INT Mars2Earth, Earth2Mars,
                    []INT FreeMemory)

  #IMPORT "earthh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [3]INT in.program:
  [3]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
    LOAD.INPUT.CHANNEL (in.program[1], ToAda)
    LOAD.INPUT.CHANNEL (in.program[2], Mars2Earth)
    LOAD.OUTPUT.CHANNEL (out.program[0], Debug)
    LOAD.OUTPUT.CHANNEL (out.program[1], FromAda)
    LOAD.OUTPUT.CHANNEL (out.program[2], Earth2Mars)
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to "earth.program".
    earth.program (ws1, in.program, out.program, dummy.ws)
:
```

This last file is required for each of the transputer due to current limitations imposed by the compiler [ALSYS 90]. The purpose of this short file is only to specify the entry point for the Ada program. This extra needed procedure is described in the Release Notes for the Alsys Ada Compilation System.

```
-- File: earthh2.occ

#OPTION "AEV"

PROC earth.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

2. Loop Harnesses

The remaining OCCAM source files for the transputers in the main loop are very close to the last two files shown in the previous section.

a. Mars

```
-- File: marsh.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC mars.harness (CHAN OF INT Mars2Earth, Earth2Mars, Venus2Mars, Mars2Pluto,
                   []INT FreeMemory)

  #IMPORT "marsh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [4]INT in.program:
  [7]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
    in.program[1] := MOSTNEG INT    -- standard i/o not used
    LOAD.INPUT.CHANNEL (in.program[2], Earth2Mars)
    LOAD.INPUT.CHANNEL (in.program[3], Venus2Mars)
    out.program[0] := MOSTNEG INT   -- standard i/o not used
    out.program[1] := MOSTNEG INT   -- standard i/o not used
    LOAD.OUTPUT.CHANNEL (out.program[2], Mars2Earth)
    out.program[3] := MOSTNEG INT   -- not used
    out.program[4] := MOSTNEG INT   -- not used
    out.program[5] := MOSTNEG INT   -- not used
    LOAD.OUTPUT.CHANNEL (out.program[6], Mars2Pluto)
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to "mars.program".
    mars.program (ws1, in.program, out.program, dummy.ws)
:

-- File: marsh2.occ

#OPTION "AEV"

PROC mars.program ([]INT ws1, in, out, ws2)
```

```
  [1000]INT d:
  SEQ
    SKIP
:
```

b. Venus

```
-- File: venush.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC venus.harness (CHAN OF INT Saturn2Venus, Venus2Mars,
                    []INT FreeMemory)

  #IMPORT "venush2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [5]INT in.program:
  [5]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT     -- not used
    in.program[1] := MOSTNEG INT     -- standard i/o not used
    in.program[2] := MOSTNEG INT     -- not used
    in.program[3] := MOSTNEG INT     -- not used
    LOAD.INPUT.CHANNEL (in.program[4], Saturn2Venus)
    out.program[0] := MOSTNEG INT   -- standard i/o not used
    out.program[1] := MOSTNEG INT   -- standard i/o not used
    out.program[2] := MOSTNEG INT   -- not used
    LOAD.OUTPUT.CHANNEL (out.program[3], Venus2Mars)
    out.program[4] := MOSTNEG INT   -- not used
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to "venus.program".
    venus.program (ws1, in.program, out.program, dummy.ws)
:

-- File: venush2.occ

#OPTION "AEV"

PROC venus.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

c. Saturn

```
-- File: saturnh.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC saturn.harness (CHAN OF INT Pluto2Saturn, Saturn2Venus,
                     []INT FreeMemory)

  #IMPORT "saturnh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [6]INT in.program:
  [5]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT     -- not used
    in.program[1] := MOSTNEG INT     -- standard i/o not used
    in.program[2] := MOSTNEG INT     -- not used
    in.program[3] := MOSTNEG INT     -- not used
    in.program[4] := MOSTNEG INT     -- not used
    LOAD.INPUT.CHANNEL (in.program[5], Pluto2Saturn)
    out.program[0] := MOSTNEG INT    -- standard i/o not used
    out.program[1] := MOSTNEG INT    -- standard i/o not used
    out.program[2] := MOSTNEG INT    -- not used
    out.program[3] := MOSTNEG INT    -- not used
    LOAD.OUTPUT.CHANNEL (out.program[4], Saturn2Venus)
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to "saturn.program".
    saturn.program (ws1, in.program, out.program, dummy.ws)
:

-- File: saturnh2.occ

#OPTION "AEV"

PROC saturn.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

d. *Pluto*

```
-- File: plutoh.occ

#OPTION "AGNVW"
#INCLUDE "hostio.inc"

PROC pluto.harness (CHAN OF INT Mars2Pluto, Pluto2Saturn,
                    []INT FreeMemory)

  #IMPORT "plutoh2.tax"

  [1]INT dummy.ws:
  ws1 IS FreeMemory:
  [7]INT in.program:
  [7]INT out.program:
  SEQ
    -- Set up vector of pointers to channels.
    in.program[0] := MOSTNEG INT    -- not used
    in.program[1] := MOSTNEG INT    -- standard i/o not used
    in.program[2] := MOSTNEG INT    -- not used
    in.program[3] := MOSTNEG INT    -- not used
    in.program[4] := MOSTNEG INT    -- not used
    in.program[5] := MOSTNEG INT    -- not used
    LOAD.INPUT.CHANNEL (in.program[6], Mars2Pluto)
    out.program[0] := MOSTNEG INT   -- standard i/o not used
    out.program[1] := MOSTNEG INT   -- standard i/o not used
    out.program[2] := MOSTNEG INT   -- not used
    out.program[3] := MOSTNEG INT   -- not used
    out.program[4] := MOSTNEG INT   -- not used
    LOAD.OUTPUT.CHANNEL (out.program[5], Pluto2Saturn)
    out.program[6] := MOSTNEG INT   -- not used
    -- Invoke the Ada program.
    -- Assumes the entry point name has been changed to "pluto.program".
    pluto.program (ws1, in.program, out.program, dummy.ws)
:

-- File: plutoh2.occ

#OPTION "AEV"

PROC pluto.program ([]INT ws1, in, out, ws2)
  [1000]INT d:
  SEQ
    SKIP
:
```

APPENDIX B: ADA SOURCE CODE

These files are the Ada source code for the simulation tasks used for this thesis. The file COMMON contains the queue size in the variable MAX_STORAGE. The Tasks QUE and WAITING are identical in all programs. The task INOUT is the same in most of the programs except for the outgoing channel name, this channel name could actually be made to be some generic name used in all programs. The one program INOUT is different is in the Mars program since it has two possible outgoing channels.

A. COMMON.ADA

```
-- File: common.ada

-- This is a common package included into all programs. Data types
-- and Channel I/O are declared here. The location procedure is
-- also included in this package.

with CHANNELS;
with CALENDAR;

package COMMON is

-- Declarations of the statistics of the network and the common
-- data types that will be used in the communication scheme.

  NUM_PROGS   : constant INTEGER := 5 ;
  NUM_PATHS   : constant INTEGER := 13;
  NUM_TASKS   : constant INTEGER := 19;
  NUM_ENTRYS  : constant INTEGER := 19;
  MAX_STORAGE : constant INTEGER := 5;
```

```
   type INT_16 is range -2**15 .. 2**15-1;
   type TASKS  is range 1..NUM_TASKS     ;
   type ENTRYS is range 1..NUM_ENTRYS    ;

   type PROG_ARRAY is array (1..NUM_PROGS) of INTEGER;
   type PATH_ARRAY is array (1..NUM_PATHS) of INTEGER;

   type PROGRAMS     is (EARTH, MARS, VENUS, SATURN, PLUTO);

-- These constants defined for passing task names in coded form
-- inside the message.  Enumeration types were used successfully
-- but increase the size of the message.  This was later found not
-- to be very important.

   SHUTDOWN          : constant TASKS := 1 ;
   HOST_TASK         : constant TASKS := 2 ;
   TASK_SCREEN       : constant TASKS := 3 ;
   EARTH_MAIN        : constant TASKS := 4 ;
   MARS_MAIN         : constant TASKS := 5 ;
   VENUS_MAIN        : constant TASKS := 6 ;
   SATURN_MAIN       : constant TASKS := 7 ;
   PLUTO_MAIN        : constant TASKS := 8 ;
   TASK_AUTO_PILOT   : constant TASKS := 9 ;
   TASK_AVOIDANCE    : constant TASKS := 10;
   TASK_EXE_MISSION  : constant TASKS := 11;
   TASK_GUIDANCE     : constant TASKS := 12;
   TASK_MONITOR      : constant TASKS := 13;
   TASK_NAVIGATION   : constant TASKS := 14;
   TASK_SONAR        : constant TASKS := 15;
   TASK_TIMER        : constant TASKS := 16;
   TASK_VEHICLE_SYS  : constant TASKS := 17;
   LOOP_TASK         : constant TASKS := 18;
   NO_TASK           : constant TASKS := 19;

-- As for the task names, the entry calls below are also assigned
-- codes for easy passing.  Again an enumeration type can be used
-- here.

   OUTPUT            : constant ENTRYS := 1 ;
   UPDATE_SONAR      : constant ENTRYS := 2 ;
   VS_ORDERS         : constant ENTRYS := 3 ;
   SYS_STATUS        : constant ENTRYS := 4 ;
   AP_ORDERS         : constant ENTRYS := 5 ;
   UPDATE_NAV        : constant ENTRYS := 6 ;
   UPDATE_ORDERS     : constant ENTRYS := 7 ;
   AVOID_REC         : constant ENTRYS := 8 ;
   SONAR_OBSTACLE    : constant ENTRYS := 9 ;
```

```
   OBJECT_ALERT     : constant ENTRYS := 10;
   EXE_UPDATE       : constant ENTRYS := 11;
   OB_AVOID         : constant ENTRYS := 12;
   MONITOR_UPDATE   : constant ENTRYS := 13;
   TO_MONITOR       : constant ENTRYS := 14;
   PILOT_UPDATE     : constant ENTRYS := 15;
   ACKNOWLEDGE      : constant ENTRYS := 16;
   NO_ENT           : constant ENTRYS := 17;
   RETURNING        : constant ENTRYS := 18;
   TEST_TIME        : constant ENTRYS := 19;

   type MESSAGE_FORM is
     record
       ORIGIN       : TASKS  := NO_TASK;
       DESTIN       : TASKS  := NO_TASK;
       ENT_CALL     : ENTRYS := NO_ENT;
       TIME_STAMP   : DURATION := 0.0;
       CODE_1       : INT_16 := 0;
       CODE_2       : INT_16 := 0;
       MESSAGE_CODE : INT_16 := 0;
       PROG         : PROG_ARRAY := (others =>0);
       PATH         : PATH_ARRAY := (others =>0);
     end record;

-- These are generic messages used in the program, shutdown is used
-- to terminate all programs.

   SHUTDOWN_MESSAGE : MESSAGE_FORM := (SHUTDOWN, SHUTDOWN, NO_ENT,
                    0.0, 0, 0, 0, (others => 0), (others => 9));
   ACK_MESSAGE      : MESSAGE_FORM := (NO_TASK, NO_TASK,
     ACKNOWLEDGE, 0.0, 0, 0, 0, (others => 0), (others => 7));

   HOST             : constant PROGRAMS := EARTH;

-- These are defined delays used during the testing of the program.
-- These are arbitrarily picked.  The value of INOUT_INT and
-- QUE_INT are the only values that appear to affect message
-- delivery time.  The given value seemed to provide the optimum
-- times, but bears further investigation.

   READ_INT         : constant DURATION := 5.0;
   SEND_INT         : constant DURATION := 0.3;
   INOUT_INT        : constant DURATION := 0.003;
   QUE_INT          : constant DURATION := 0.003;
   AVOIDANCE_INT    : constant DURATION := 0.08;
   PILOT_INT        : constant DURATION := 0.04;
   SONAR_INT        : constant DURATION := 0.02;
```

```
  VEHICLE_INT    : constant DURATION := 0.08;
  MONITOR_INT    : constant DURATION := 0.03;
  EXE_INT        : constant DURATION := 0.06;
  GUIDANCE_INT   : constant DURATION := 0.07;
  NAVIGATION_INT : constant DURATION := 0.05;

-- Instantiations of the generic channel i/o package.

  package MESSAGE_IO is new CHANNELS.CHANNEL_IO (MESSAGE_FORM);

  function IF_ITS_HERE (FROM_PROGRAM : in PROGRAMS; TO_TASK : in
         TASKS)
    return BOOLEAN;

end COMMON;


package body COMMON is

  function IF_ITS_HERE (FROM_PROGRAM : in PROGRAMS; TO_TASK : in
         TASKS)
  return BOOLEAN is

  ANSWER : BOOLEAN := FALSE;

  begin
    case FROM_PROGRAM is
      when EARTH  =>
        if (TO_TASK = HOST_TASK)        or
          (TO_TASK = TASK_SCREEN)      then ANSWER := TRUE;
        end if;
      when MARS   =>
        if (TO_TASK = TASK_AUTO_PILOT)  or
          (TO_TASK = TASK_VEHICLE_SYS) or
          (TO_TASK = TASK_TIMER)       then ANSWER := TRUE;
        end if;
      when VENUS  =>
        if (TO_TASK = TASK_NAVIGATION)  or
          (TO_TASK = TASK_SONAR)       then ANSWER := TRUE;
        end if;
      when SATURN =>
        if (TO_TASK = TASK_AVOIDANCE)   or
          (TO_TASK = TASK_GUIDANCE)    then ANSWER := TRUE;
        end if;
      when PLUTO  =>
        if (TO_TASK = LOOP_TASK)        or
          (TO_TASK = TASK_EXE_MISSION) or
```

```
            (TO_TASK = TASK_MONITOR)      then ANSWER := TRUE;
         end if;
      when others     =>
         ANSWER := FALSE;
   end case;

   return ANSWER;

  end IF_ITS_HERE;

end COMMON;
```

B. PRINTOUT.ADA

This package was used simply for formatted output when it was desired to print out the entire message. This file was only included in the EARTH program.

```
-- File: printout.ada

with COMMON;
with TEXT_IO;
package PRINTOUT is

  use COMMON;

  package PRINT_TASK  is new TEXT_IO.INTEGER_IO (TASKS)    ;
  package PRINT_PROG  is new TEXT_IO.ENUMERATION_IO (PROGRAMS);
  package INT_IO is new TEXT_IO.INTEGER_IO(INT_16)          ;

  procedure PRINT_MESSAGE (MESSAGE : in MESSAGE_FORM);

end PRINTOUT;


package body PRINTOUT is

  procedure PRINT_MESSAGE (MESSAGE : in MESSAGE_FORM) is

    TO_TASK_NAME   : TASKS  ;
    FROM_TASK_NAME : TASKS  ;
    I              : INTEGER;
```

```
  begin

     FROM_TASK_NAME := MESSAGE.ORIGIN     ;

     TEXT_IO.NEW_LINE;
     TEXT_IO.PUT_LINE
("*******************************************************");
     TEXT_IO.PUT_LINE
("*                 Message Report                    *");
     TEXT_IO.PUT_LINE
("*                                                   *");
     TEXT_IO.PUT     ("*  From       : ");
     PRINT_TASK.PUT (FROM_TASK_NAME,30) ;
     TEXT_IO.PUT_LINE ("         *");
     TEXT_IO.PUT_LINE
("*                   Path Array:                     *");
     TEXT_IO.PUT ("*              ");

     for I in 1..NUM_PATHS loop
        TEXT_IO.PUT (" ");
        INT_IO.PUT (INT_16 (MESSAGE.PATH(I)),3);
     end loop;
     TEXT_IO.PUT_LINE ("             *");
     TEXT_IO.PUT_LINE
("*                 Program Array:                    *");
     TEXT_IO.PUT ("*               ");

     for I in 1..NUM_PROGS loop
        TEXT_IO.PUT (" ");
        INT_IO.PUT  (INT_16 (MESSAGE.PROG(I)),3);
        TEXT_IO.PUT (" ");
     end loop;

     TEXT_IO.PUT_LINE ("               *");
     TEXT_IO.PUT      ("* CODE_1 : ");
     INT_IO.PUT (MESSAGE.CODE_1,3);
     TEXT_IO.PUT      ("   CODE_2 : ");
     INT_IO.PUT (MESSAGE.CODE_2,3);
     TEXT_IO.PUT      ("  Message Code : ");
     INT_IO.PUT (MESSAGE.MESSAGE_CODE);
     TEXT_IO.PUT_LINE ("  *");
     TEXT_IO.PUT_LINE
("*******************************************************");

  end PRINT_MESSAGE;

end PRINTOUT;
```

C. EARTH.ADA

```
-- File: earth.ada
-- Author: Clay Richmond
--
-- Tasks included in this program:              Entry calls
--
--     QUE                                 TO_QUE
--     SCREEN                                OUTPUT
--

with COMMON;
with PRINTOUT;
with TEXT_IO;
with CHANNELS;
with CALENDAR;

procedure EARTH is

   use COMMON;
   use PRINTOUT;
   use CALENDAR;

   package TIME_IO is new TEXT_IO.FIXED_IO (DURATION);
   package FLT_IO  is new TEXT_IO.FLOAT_IO (FLOAT);

   IN_MESSAGE   : MESSAGE_FORM             ;
   MAIN_TALK    : MESSAGE_FORM             ;
   LOCATION    : constant PROGRAMS := EARTH;
   TIME_OUT    : TIME;
   QUIT_TIME   : TIME;
   ABORTED     : BOOLEAN;
   FAILED     : INT_16 := 0;
   MESS_COUNT   : INT_16 := 0;
   QUIT_INT    : DURATION := 50.0;

   task QUE is
     entry TO_QUE        (QUE_MESSAGE : in MESSAGE_FORM)   ;
   end;

   task SCREEN is
     entry OUTPUT        (SCREEN_MESSAGE : in MESSAGE_FORM);
   end;

   InFromMars : CHANNELS.CHANNEL_REF := CHANNELS.IN_PARAMETERS (2);
```

```
OutToMars  : CHANNELS.CHANNEL_REF := CHANNELS.OUT_PARAMETERS(2);

task body QUE is

   SENT_MESSAGE     : BOOLEAN          := FALSE;
   SENT_ACK         : BOOLEAN          := FALSE;
   ALL_FULL         : BOOLEAN          := FALSE;
   FULL             : constant BOOLEAN := TRUE;
   EMPTY            : constant BOOLEAN := FALSE;
   NUMBER           : INTEGER          := 0;
   MESSAGES_IN_MAIL : INTEGER          := 0;
   SLOT             : array(0 .. (MAX_STORAGE-1)) of BOOLEAN :=
                                        (others => FALSE);
   STORAGE          : array(0 .. (MAX_STORAGE-1)) of
                                        MESSAGE_FORM;
   TEMP_MESSAGE     : MESSAGE_FORM;
   TALK             : MESSAGE_FORM;

   procedure SEND_IT (MESSAGE : in MESSAGE_FORM;
                       ACK : out BOOLEAN;
                      MESS : out BOOLEAN) is

      MESSAGE_SENT : BOOLEAN := FALSE;
      ACK_SENT     : BOOLEAN := FALSE;

    begin
      select
        SCREEN.OUTPUT (STORAGE (NUMBER));
        MESSAGE_SENT := TRUE;
      or
        delay 0.01;
      end select;

      ACK  := ACK_SENT;
      MESS := MESSAGE_SENT;
      return;
    end SEND_IT;

begin

   MAIN: loop
    select
      accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM) do
        TEMP_MESSAGE := QUE_MESSAGE;
      end TO_QUE;

      STORAGE (NUMBER) := TEMP_MESSAGE;
```

```
          MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
          SLOT (NUMBER) := FULL;

          SEND: loop
            if ALL_FULL = FALSE then
              select
                accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM)
                  do
                  TEMP_MESSAGE := QUE_MESSAGE;
                end TO_QUE;

                if MESSAGES_IN_MAIL < MAX_STORAGE then
                  STORE: loop
                    if SLOT (NUMBER) = EMPTY then
                      STORAGE (NUMBER) := TEMP_MESSAGE;
                      MESSAGES_IN_MAIL:=MESSAGES_IN_MAIL+1;
                      SLOT (NUMBER) := FULL;
                      exit;
                    end if;

-- Add 1 to NUMBER so that next mail slot can be checked.

                    NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
                  end loop STORE;

-- Add 1 to NUMBER so that last in will not be first out if there
-- are other messages in the queue.

                  NUMBER := (NUMBER + 1) MOD MAX_STORAGE;

-- This is a flag that says that are incoming messages not yet
-- stored in the queue, and no others should be read until it is.

                else
                  ALL_FULL := TRUE;
                end if;
              or
                delay 0.0;
              end select;
            end if;

-- Priority is given to any messages waiting to be mailed, so
-- another ACCEPT statement is needed before attempting to deliver
-- a message.

            if SLOT (NUMBER) = FULL then
              SEND_IT (STORAGE (NUMBER), SENT_ACK,
```

```
                                    SENT_MESSAGE);
          end if;

          if SENT_MESSAGE then
            SENT_MESSAGE := FALSE;
            SLOT (NUMBER) := EMPTY;
            MESSAGES_IN_MAIL := MESSAGES_IN_MAIL - 1;
            if ALL_FULL then
              STORAGE (NUMBER) := TEMP_MESSAGE;
              SLOT (NUMBER)    := FULL;
              MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
              ALL_FULL := FALSE;
            end if;
          end if;

          NUMBER := (NUMBER + 1) MOD MAX_STORAGE;

-- To save processor time the loop is exited when there are no
-- pending mail deliveries.

          exit when MESSAGES_IN_MAIL = 0;
        end loop SEND;
      or
        terminate;
      end select;
    end loop MAIN;

  end QUE;

  task body SCREEN is

    OUT2SCREEN : MESSAGE_FORM;
    LOCALS     : array (TASKS) of INT_16 := (others => 0);
    COUNT      : INTEGER := 0;
    N          : INTEGER := 0;
    AVE_TIME   : FLOAT;
    START_STAMP: CALENDAR.TIME;
    TIMER      : DURATION := 0.0;
    TOT_TIME   : DURATION := 0.0;
    OUT_TIME   : DURATION := 0.0;

  begin

    MAIN: loop
        accept OUTPUT (SCREEN_MESSAGE : in MESSAGE_FORM) do
          OUT2SCREEN := SCREEN_MESSAGE;
        end OUTPUT;
```

```
-- This case statement lists all the message codes with the
-- associated messages.  This of course can be expanded to include
-- any necessary correspondence with the screen.

        case OUT2SCREEN.MESSAGE_CODE is
           when 11 =>
              TEXT_IO.PUT_LINE
                   ("Main Earth program finished.");
              PRINT_MESSAGE(OUT2SCREEN);
              exit;
           when 20 =>
              LOCALS (OUT2SCREEN.ORIGIN) :=
                       LOCALS (OUT2SCREEN.ORIGIN) + 1;
              TOT_TIME := TOT_TIME + OUT2SCREEN.TIME_STAMP;
              N := N + 1;
           when 21 =>
              LOCALS (OUT2SCREEN.ORIGIN) :=
                       LOCALS (OUT2SCREEN.ORIGIN) + 1;
           when 30 =>
              START_STAMP := CLOCK;
           when 31 =>
              TIMER := CLOCK - START_STAMP;
              OUT_TIME := OUT2SCREEN.TIME_STAMP;
           when 99 =>
              TEXT_IO.PUT_LINE ("Shutdown Received.");
           when others =>
              TEXT_IO.PUT_LINE ("Bad MESSAGE_CODE.");
              PRINT_MESSAGE(OUT2SCREEN);
        end case;
     end loop MAIN;

     TEXT_IO.PUT ("EARTH_MAIN = ");
     INT_IO.PUT (LOCALS(EARTH_MAIN));
     TEXT_IO.NEW_LINE;

     TEXT_IO.PUT ("TASK_VEHICLE_SYS = ");
     INT_IO.PUT (LOCALS(TASK_VEHICLE_SYS));
     TEXT_IO.NEW_LINE;

     TEXT_IO.PUT ("Total time from SCREEN was : ");
     TIME_IO.PUT (TIMER);
     TEXT_IO.NEW_LINE;

     TEXT_IO.PUT ("Total time from VEHICLE_SYS was : ");
     TIME_IO.PUT (OUT_TIME);
     TEXT_IO.NEW_LINE;
```

```
      TEXT_IO.PUT ("Ave Time calculated from VEHICLE_SYS was : ");
      AVE_TIME := FLOAT(TOT_TIME) / FLOAT(N);
      FLT_IO.PUT (AVE_TIME);
      TEXT_IO.NEW_LINE;

   end SCREEN;

begin

   QUIT_TIME := CLOCK + QUIT_INT;

   MAIN_TALK.DESTIN        := TASK_SCREEN;
   MAIN_TALK.ORIGIN        := EARTH_MAIN;
   MAIN_TALK.MESSAGE_CODE := 21;
   QUE.TO_QUE (MAIN_TALK);

   loop
      TIME_OUT := CLOCK + READ_INT;
      MESSAGE_IO.READ_OR_FAIL (InFromMars, IN_MESSAGE, TIME_OUT,
                              ABORTED);

      if ABORTED then
         FAILED := FAILED + 1;
      else
         MESS_COUNT := MESS_COUNT + 1;
         IN_MESSAGE.PROG(1) := IN_MESSAGE.PROG(1) + 1;
         QUE.TO_QUE (IN_MESSAGE);
      end if;

      exit when CLOCK > QUIT_TIME;
   end loop;

   delay 0.5;

   MAIN_TALK := IN_MESSAGE;
   MAIN_TALK.DESTIN        := TASK_SCREEN;
   MAIN_TALK.MESSAGE_CODE := 11;
   MAIN_TALK.CODE_1 := FAILED;
   MAIN_TALK.CODE_2 := MESS_COUNT;
   QUE.TO_QUE (MAIN_TALK);

end EARTH;
```

D. MARS.ADA

```
-- File: mars.ada
-- Author: Clay Richmond
--
-- Tasks included in this program:          Entry calls
--
--     INOUT                               INCOMING, SEND
--     QUE                                 TO_QUE
--     AUTO_PILOT                            AP_ORDERS, PILOT_UPDATE, ACK
--     TIMER                               TEST_TIME, ACK
--     VEHICLE_SYS                           VS_ORDERS,GO, FIN, ACK
--

with COMMON;
with CHANNELS;
with CALENDAR;

procedure MARS is

  use COMMON;
  use CALENDAR;

  IN_MESSAGE     : MESSAGE_FORM;
  LOCATION       : constant PROGRAMS   :=  MARS;
  STOPPER        : constant INTEGER := 100;

  task INOUT is
    entry INCOMING (INOUT_MESSAGE : in MESSAGE_FORM);
    entry SEND     (INOUT_MESSAGE : in MESSAGE_FORM);
  end;

  task WAITING is
    entry LIMBO (WAIT_MESSAGE : in MESSAGE_FORM);
  end;

  task QUE is
    entry TO_QUE (QUE_MESSAGE : in MESSAGE_FORM);
  end;

  task AUTO_PILOT is
    entry AP_ORDERS    (PILOT_MESSAGE : in MESSAGE_FORM);
    entry PILOT_UPDATE (PILOT_MESSAGE : in MESSAGE_FORM);
    entry ACK;
  end;
```

```
task TIMER is
   entry TEST_TIME;
   entry ACK;
end;

task VEHICLE_SYS is
   entry VS_ORDERS    (VS_MESSAGE : in MESSAGE_FORM)    ;
   entry GO                                             ;
   entry FIN                                            ;
   entry ACK                                            ;
end;

OutToEarth  : CHANNELS.CHANNEL_REF:= CHANNELS.OUT_PARAMETERS(2);
InFromEarth : CHANNELS.CHANNEL_REF:= CHANNELS.IN_PARAMETERS (2);
InFromVenus : CHANNELS.CHANNEL_REF:= CHANNELS.IN_PARAMETERS (3);
OutToPluto  : CHANNELS.CHANNEL_REF:= CHANNELS.OUT_PARAMETERS(6);

procedure SEND_IT (MESSAGE : in MESSAGE_FORM;
                   ACK : out BOOLEAN;
                  MESS : out BOOLEAN) is

   MESSAGE_SENT : BOOLEAN := FALSE;
   ACK_SENT     : BOOLEAN := FALSE;

begin
   case MESSAGE.DESTIN is
      when TASK_AUTO_PILOT =>
         case MESSAGE.ENT_CALL is
            when AP_ORDERS =>
               select
                  AUTO_PILOT.AP_ORDERS (MESSAGE);
                  MESSAGE_SENT := TRUE;
               or
                  delay 0.0;
               end select;
            when PILOT_UPDATE =>
               select
                  AUTO_PILOT.PILOT_UPDATE (MESSAGE);
                  MESSAGE_SENT := TRUE;
               or
                  delay 0.0;
               end select;
            when ACKNOWLEDGE =>
               select
                  AUTO_PILOT.ACK;
                  ACK_SENT := TRUE;
               or
```

```
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
      end case;
    when TASK_TIMER =>
      case MESSAGE.ENT_CALL is
        when TEST_TIME =>
          select
            TIMER.TEST_TIME;
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when ACKNOWLEDGE =>
          select
            TIMER.ACK;
            ACK_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
      end case;
    when TASK_VEHICLE_SYS  =>
      case MESSAGE.ENT_CALL is
        when VS_ORDERS =>
          select
            VEHICLE_SYS.VS_ORDERS (MESSAGE);
            MESSAGE_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when ACKNOWLEDGE =>
          select
            VEHICLE_SYS.ACK;
            ACK_SENT := TRUE;
          or
            delay 0.0;
          end select;
        when others => null;  -- Not a valid call
      end case;
    when others  => null;  -- not a valid task
  end case;

  ACK  := ACK_SENT;
  MESS := MESSAGE_SENT;
  return;
end SEND_IT;
```

```
task body INOUT is

   COUNT              : INTEGER := 0;
   HERE               : BOOLEAN;
   ON_HOST            : BOOLEAN;
   ABORTED            : BOOLEAN;
   STORE_MESSAGE      : MESSAGE_FORM;
   TIME_OUT           : CALENDAR.TIME ;

begin

   loop
      select
         accept INCOMING (INOUT_MESSAGE : in MESSAGE_FORM) do
            STORE_MESSAGE := INOUT_MESSAGE;
         end INCOMING;
      or
         accept SEND (INOUT_MESSAGE : in MESSAGE_FORM) do
            STORE_MESSAGE := INOUT_MESSAGE;
         end SEND;
      or
         terminate;
      end select;

      ON_HOST := IF_ITS_HERE (HOST, STORE_MESSAGE.DESTIN);
      HERE    := IF_ITS_HERE (LOCATION, STORE_MESSAGE.DESTIN);

      TIME_OUT := CLOCK + INOUT_INT;

      if ON_HOST then
         MESSAGE_IO.WRITE_OR_FAIL (OutToEarth, STORE_MESSAGE,
                  TIME_OUT, ABORTED);
         if ABORTED then
            MESSAGE_IO.WRITE (OutToPluto, STORE_MESSAGE);
         end if;
      elsif HERE then
         select
            QUE.TO_QUE (STORE_MESSAGE);
         or
            delay INOUT_INT;
         end select;
      else
         MESSAGE_IO.WRITE (OutToPluto, STORE_MESSAGE);
      end if;

   end loop;
end INOUT;
```

```
task body WAITING is

   MAX_STORAGE : INTEGER := 5;
   BOTTOM      : INTEGER := 0;
   TOP         : INTEGER := 0;
   LIST        : array (0..(MAX_STORAGE-1)) of MESSAGE_FORM;

begin
   loop
      select
         accept LIMBO (WAIT_MESSAGE : in MESSAGE_FORM) do
            LIST (TOP) := WAIT_MESSAGE;
         end LIMBO;
         TOP := (TOP + 1) MOD MAX_STORAGE;

         loop
            select
               accept LIMBO (WAIT_MESSAGE : in MESSAGE_FORM) do
                  LIST (TOP) := WAIT_MESSAGE;
               end LIMBO;
               TOP := (TOP + 1) MOD MAX_STORAGE;
            else
               select
                  INOUT.SEND (LIST (BOTTOM));
                  BOTTOM := (BOTTOM + 1) MOD MAX_STORAGE;
                  exit when BOTTOM = TOP;
               or
                  delay 0.02;
               end select;
            end select;
         end loop;
      or
         terminate;
      end select;
   end loop;
end WAITING;

task body QUE is

   SENT_MESSAGE     : BOOLEAN          := FALSE;
   SENT_ACK         : BOOLEAN          := FALSE;
   ALL_FULL         : BOOLEAN          := FALSE;
   FULL             : constant BOOLEAN := TRUE;
   EMPTY            : constant BOOLEAN := FALSE;
   NUMBER           : INTEGER          := 0;
   MESSAGES_IN_MAIL : INTEGER          := 0;
   SLOT             : array(0 .. (MAX_STORAGE-1)) of BOOLEAN :=
```

```
                                        (others => FALSE);
   STORAGE          : array(0 .. (MAX_STORAGE-1)) of
                                        MESSAGE_FORM;
   TEMP_MESSAGE     : MESSAGE_FORM;

 begin

   MAIN: loop
     select
       accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM) do
         TEMP_MESSAGE := QUE_MESSAGE;
       end TO_QUE;

       STORAGE (NUMBER) := TEMP_MESSAGE;
       MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
       SLOT (NUMBER) := FULL;

-- Priority is given to any messages waiting to be mailed, so
-- another ACCEPT statement is needed before attempting to deliver
-- a message.

       SEND: loop
         if ALL_FULL = FALSE then
           select
             accept TO_QUE (QUE_MESSAGE : in MESSAGE_FORM)
               do
               TEMP_MESSAGE := QUE_MESSAGE;
             end TO_QUE;

             if MESSAGES_IN_MAIL < MAX_STORAGE then
               STORE: loop
                 if SLOT (NUMBER) = EMPTY then
                   STORAGE (NUMBER) := TEMP_MESSAGE;
                   MESSAGES_IN_MAIL:=MESSAGES_IN_MAIL+1;
                   SLOT (NUMBER) := FULL;
                   exit;
                 end if;

-- Add 1 to NUMBER so that next mail slot can be checked.

                 NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
               end loop STORE;

-- Add 1 to NUMBER so that in the SEND loop the last mail slot
-- filled will not be the first to be checked for sending.

               NUMBER := (NUMBER + 1) MOD MAX_STORAGE;
```

-- This is a flag that says that are incoming messages not yet
-- stored in the queue, and no others should be read until it is.

```
               else
                  ALL_FULL := TRUE;
               end if;
            or
               delay 0.0;
            end select;
         end if;

         if SLOT (NUMBER) = FULL then
            SEND_IT (STORAGE (NUMBER), SENT_ACK,
                                 SENT_MESSAGE);
         end if;

         if SENT_MESSAGE then
            SENT_MESSAGE := FALSE;
            ACK_MESSAGE.DESTIN    := STORAGE (NUMBER).ORIGIN;
            ACK_MESSAGE.ORIGIN    := STORAGE (NUMBER).DESTIN;
            SLOT (NUMBER) := EMPTY;
            MESSAGES_IN_MAIL := MESSAGES_IN_MAIL - 1;
            if ALL_FULL then
               STORAGE (NUMBER) := TEMP_MESSAGE;
               SLOT (NUMBER)    := FULL;
               MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
               ALL_FULL := FALSE;
            end if;
```

-- Now the acknowledgement message is sent, but if INOUT is trying
-- to mail a message deadlock will occur. So again incoming
-- messages is given priority.

```
            SEND_ACK: loop
               if ALL_FULL = FALSE then
                  select
                     accept TO_QUE (QUE_MESSAGE : in
      MESSAGE_FORM) do
                        TEMP_MESSAGE := QUE_MESSAGE;
                     end TO_QUE;

                     if MESSAGES_IN_MAIL < MAX_STORAGE then
                        NEXT_STORE: loop
                           if SLOT (NUMBER) = EMPTY then
                              STORAGE (NUMBER):=TEMP_MESSAGE;
                              MESSAGES_IN_MAIL:=
                                   MESSAGES_IN_MAIL + 1;
```

```
                    SLOT (NUMBER) := FULL;
                    exit;
                  end if;
                  NUMBER :=
                     (NUMBER + 1) MOD MAX_STORAGE;
                end loop NEXT_STORE;
                NUMBER :=
                     (NUMBER + 1) MOD MAX_STORAGE;
              else
                ALL_FULL := TRUE;
              end if;
            or
              delay 0.0;
            end select;
          end if;

          select
            INOUT.SEND (ACK_MESSAGE);
            exit;
          or
            delay QUE_INT;
          end select;
        end loop SEND_ACK;
      elsif SENT_ACK then
        SENT_ACK              := FALSE;
        MESSAGES_IN_MAIL      := MESSAGES_IN_MAIL - 1;
        SLOT (NUMBER)         := EMPTY;
        if ALL_FULL then
          STORAGE (NUMBER) := TEMP_MESSAGE;
          SLOT (NUMBER)    := FULL;
          MESSAGES_IN_MAIL := MESSAGES_IN_MAIL + 1;
          ALL_FULL := FALSE;
        end if;
      end if;

-- If a message was not sent and neither was an acknowledgement,
-- then the receiving task was not ready.  So 1 is added to NUMBER
-- and the next message found will get its chance at being
-- delivered.

      NUMBER := (NUMBER + 1) MOD MAX_STORAGE;

-- To save processor time the loop is exited when there are no
-- pending mail deliveries.

      exit when MESSAGES_IN_MAIL = 0;
    end loop SEND;
```

```
      or
        terminate;
      end select;
    end loop MAIN;

  end QUE;

  task body AUTO_PILOT is

-- This task receives its input from GUIDANCE, it then waits for an
-- update from VEHICLE_SYS before then sending its signals back to
-- VEHICLE_SYS. For the purposes of this dummy task, no decisions
-- are made here.

    IN_TASK, NEW_SYS : MESSAGE_FORM;

  begin

    loop
      select
        accept PILOT_UPDATE (PILOT_MESSAGE : in MESSAGE_FORM)          do
          NEW_SYS := PILOT_MESSAGE;
        end PILOT_UPDATE;

        accept AP_ORDERS (PILOT_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := PILOT_MESSAGE;
        end AP_ORDERS;

        delay PILOT_INT;

        IN_TASK.ORIGIN    := TASK_AUTO_PILOT;
        IN_TASK.DESTIN    := TASK_VEHICLE_SYS;
        IN_TASK.ENT_CALL  := VS_ORDERS;
        INOUT.SEND (IN_TASK);

        accept ACK;
      or
        terminate;
      end select;
    end loop;
  end AUTO_PILOT;

  task body TIMER is

    COUNT   : INTEGER        := 0 ;
    TALK : MESSAGE_FORM;
```

```
  begin
      delay 0.3;
-- this delay allows initialization to complete.

    loop
      VEHICLE_SYS.GO;
      COUNT := COUNT + 1;
      delay 0.25;
-- this is the delay that controls frequency.

      exit when COUNT = STOPPER;
    end loop;

    delay 3.0;
-- this delay enables all tasks to come to completion

-- Shutdown message is sent to the last program in the loop
-- and then returns to EARTH.  On its way to earth all programs
-- terminate as it passes.

    TALK              := SHUTDOWN_MESSAGE;
    TALK.DESTIN       := LOOP_TASK;
    TALK.MESSAGE_CODE := 99;
    INOUT.SEND (TALK);
  end TIMER;

  task body VEHICLE_SYS is

-- This task receives from AUTO_PILOT and interfaces with the
-- external vehicle systems which are not shown in this dummy task.
-- Output is sent to SONAR and NAVIGATION.  The first four messages
-- sent is the initialization.  It was found that arranging the
-- placement of the accept statements for the acknowledgment
-- message had an impact on timing.

    IN_TASK, TALK   : MESSAGE_FORM;
    NEXT_TIME   : CALENDAR.TIME := CLOCK + VEHICLE_INT;
    PRE_STAMP   : CALENDAR.TIME;
    START_STAMP : CALENDAR.TIME;
    TIMER       : DURATION := 0.0;
    FINAL       : DURATION := 0.0;
    COUNT       : INTEGER  := 0;

  begin

    PRE_STAMP := CLOCK;
```

```
-- All messages in this task sent to SCREEN were for data
-- collection purposes only.

    TALK.ORIGIN       := TASK_VEHICLE_SYS;
    TALK.DESTIN       := TASK_SCREEN    ;
    TALK.ENT_CALL     := OUTPUT         ;
    TALK.MESSAGE_CODE := 30             ;
    INOUT.SEND (TALK);

    IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
    IN_TASK.DESTIN      := TASK_NAVIGATION ;
    IN_TASK.ENT_CALL    := SYS_STATUS      ;

    INOUT.SEND (IN_TASK);

    IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
    IN_TASK.DESTIN      := TASK_SONAR      ;
    IN_TASK.ENT_CALL    := UPDATE_SONAR    ;

    INOUT.SEND (IN_TASK);

    IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
    IN_TASK.DESTIN      := TASK_MONITOR    ;
    IN_TASK.ENT_CALL    := TO_MONITOR      ;

    INOUT.SEND (IN_TASK);

    IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
    IN_TASK.DESTIN      := TASK_AUTO_PILOT ;
    IN_TASK.ENT_CALL    := PILOT_UPDATE    ;

    INOUT.SEND (IN_TASK);
    accept ACK;
    accept ACK;
    accept ACK;
    accept ACK;

    loop
      select
        accept GO;

        START_STAMP := CLOCK;

        accept VS_ORDERS (VS_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := VS_MESSAGE;
        end VS_ORDERS;
```

```
        delay VEHICLE_INT;

        IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
        IN_TASK.DESTIN      := TASK_NAVIGATION ;
        IN_TASK.ENT_CALL    := SYS_STATUS      ;

        INOUT.SEND (IN_TASK);

        IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
        IN_TASK.DESTIN      := TASK_SONAR      ;
        IN_TASK.ENT_CALL    := UPDATE_SONAR    ;

        INOUT.SEND (IN_TASK);
        accept ACK;
        accept ACK;

        IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
        IN_TASK.DESTIN      := TASK_MONITOR    ;
        IN_TASK.ENT_CALL    := TO_MONITOR      ;

        INOUT.SEND (IN_TASK);

        IN_TASK.ORIGIN      := TASK_VEHICLE_SYS;
        IN_TASK.DESTIN      := TASK_AUTO_PILOT ;
        IN_TASK.ENT_CALL    := PILOT_UPDATE    ;

        INOUT.SEND (IN_TASK);
        accept ACK;
        accept ACK;

-- TIMER is the loop iteration time, it does not time interval
-- between iterations.

        TIMER := CLOCK - START_STAMP;

        TALK.ORIGIN        := TASK_VEHICLE_SYS;
        TALK.DESTIN        := TASK_SCREEN     ;
        TALK.ENT_CALL      := OUTPUT          ;
        TALK.TIME_STAMP    := TIMER           ;
        TALK.MESSAGE_CODE  := 20              ;
        INOUT.SEND (TALK);

        COUNT := COUNT + 1;

        exit when COUNT = STOPPER;
      or
        accept FIN;
```

```
        exit;
      end select;
    end loop;

    accept VS_ORDERS (VS_MESSAGE : in MESSAGE_FORM) do
      IN_TASK := VS_MESSAGE;
    end VS_ORDERS;

    FINAL := CLOCK - PRE_STAMP;

    TALK.ORIGIN        := TASK_VEHICLE_SYS;
    TALK.DESTIN        := TASK_SCREEN      ;
    TALK.ENT_CALL      := OUTPUT           ;
    TALK.TIME_STAMP    := FINAL            ;
    TALK.MESSAGE_CODE := 31                ;
    INOUT.SEND (TALK);

  end VEHICLE_SYS;

begin

  loop
    MESSAGE_IO.READ (InFromVenus, IN_MESSAGE);

    IN_MESSAGE.PROG(2) := IN_MESSAGE.PROG(2) + 1;

    if IN_MESSAGE.ORIGIN = SHUTDOWN  and IN_MESSAGE.DESTIN =
                         HOST_TASK then
      IN_MESSAGE.PROG(2) := -1 * IN_MESSAGE.PROG(2);
      delay 1.0;
      INOUT.INCOMING (IN_MESSAGE);
      exit;
    end if;

    select
      INOUT.INCOMING (IN_MESSAGE);
    or
      delay SEND_INT;
      WAITING.LIMBO (IN_MESSAGE);
    end select;
  end loop;

end MARS;
```

E. TASKS

The remaining programs include VENUS.ADA, SATURN.ADA, and PLUTO.ADA. These programs are all simple variations to the above programs, and therefore, are not included here. All time measurements were taken with the task delays set at zero rather than the arbitrary numbers given above. The simulation tasks, not provided above, are as follows:

```
  task NAVIGATION is
    entry SONAR_OBSTACLE (NAV_MESSAGE : in MESSAGE_FORM);
    entry SYS_STATUS     (NAV_MESSAGE : in MESSAGE_FORM);
    entry ACK;
  end;

  task body NAVIGATION is

-- This task takes the output from either SONAR or it will accept an update from
-- external systems (VEHICLE_SYS). Output is then sent to GUIDANCE in either
-- case.

    IN_TASK   : MESSAGE_FORM;

  begin

    loop
      select
        accept SYS_STATUS (NAV_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := NAV_MESSAGE;
        end SYS_STATUS;

        accept SONAR_OBSTACLE (NAV_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := NAV_MESSAGE;
        end SONAR_OBSTACLE;

        delay NAVIGATION_INT;

        IN_TASK.ORIGIN     := TASK_NAVIGATION;
        IN_TASK.DESTIN     := TASK_GUIDANCE;
        IN_TASK.ENT_CALL   := UPDATE_NAV;

        INOUT.SEND (IN_TASK);
        accept ACK;
```

```
        or
          terminate;
        end select;
      end loop;

    end NAVIGATION;

    task SONAR is
      entry UPDATE_SONAR (SONAR_MESSAGE : in MESSAGE_FORM) ;
      entry ACK                                            ;
    end;

    task body SONAR is

-- This task receives from VEHICLE_SYS  and raw sonar data from the external
-- sensors.  It then sends processes data to NAVIGATION.  If an object is
-- detected in a danger area then the information is sent to AVOIDANCE.  In
-- this dummy task, a message was always sent.

      IN_TASK         : MESSAGE_FORM ;
      EMERG_MESSAGE   : MESSAGE_FORM ;

    begin

      loop
        select

-- Awaits data from VEHICLE_SYS task.

          accept UPDATE_SONAR (SONAR_MESSAGE : in MESSAGE_FORM) do
            IN_TASK := SONAR_MESSAGE;
          end UPDATE_SONAR;

          EMERG_MESSAGE.ORIGIN    := TASK_SONAR    ;
          EMERG_MESSAGE.DESTIN    := TASK_AVOIDANCE;
          EMERG_MESSAGE.ENT_CALL  := OB_AVOID      ;
          INOUT.SEND (EMERG_MESSAGE)               ;
          accept ACK;

          IN_TASK.ORIGIN    := TASK_SONAR     ;
          IN_TASK.DESTIN    := TASK_NAVIGATION;
          IN_TASK.ENT_CALL  := SONAR_OBSTACLE ;
          INOUT.SEND (IN_TASK);
          accept ACK;

          delay SONAR_INT;
        or
```

```
        terminate;
      end select;
    end loop;

  end SONAR;

  task AVOIDANCE is
    entry OB_AVOID (AVOID_MESSAGE : in MESSAGE_FORM);
    entry ACK;
  end;

  task body AVOIDANCE is

-- This task receives only from SONAR and at irregular intervals.  When input
-- is received, output is sent to both EXE_MISSION and GUIDANCE.

    IN_TASK : MESSAGE_FORM;

  begin

    loop
      select
        accept OB_AVOID (AVOID_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := AVOID_MESSAGE;
        end OB_AVOID;

        IN_TASK.ORIGIN     := TASK_AVOIDANCE ;
        IN_TASK.DESTIN     := TASK_GUIDANCE  ;
        IN_TASK.ENT_CALL   := AVOID_REC      ;
        INOUT.SEND (IN_TASK);
        accept ACK;

        delay AVOIDANCE_INT;

        IN_TASK.ORIGIN     := TASK_AVOIDANCE  ;
        IN_TASK.DESTIN     := TASK_EXE_MISSION;
        IN_TASK.ENT_CALL   := OBJECT_ALERT    ;
        INOUT.SEND (IN_TASK);
        accept ACK;
      or
        terminate;
      end select;
    end loop;

  end AVOIDANCE;

  task GUIDANCE is
```

```
    entry UPDATE_NAV     (GUIDE_MESSAGE : in MESSAGE_FORM);
    entry UPDATE_ORDERS (GUIDE_MESSAGE : in MESSAGE_FORM);
    entry AVOID_REC      (GUIDE_MESSAGE : in MESSAGE_FORM);
    entry ACK ;
  end;

  task body GUIDANCE is

-- This task receives input regularly from NAVIGATION and EXE_MISSION.  Plus
-- it receives input from AVOIDANCE irregularly.  Output is always sent
-- to AUTO_PILOT.

    EMERG, IN_TASK, GO_TO, WE_ARE : MESSAGE_FORM;

  begin

    loop
      select
        accept UPDATE_NAV (GUIDE_MESSAGE : in MESSAGE_FORM) do
          WE_ARE := GUIDE_MESSAGE;
        end UPDATE_NAV;

        accept AVOID_REC (GUIDE_MESSAGE : in MESSAGE_FORM) do
          EMERG := GUIDE_MESSAGE;
        end AVOID_REC;

        accept UPDATE_ORDERS (GUIDE_MESSAGE : in MESSAGE_FORM) do
          GO_TO := GUIDE_MESSAGE;
        end UPDATE_ORDERS;

        delay GUIDANCE_INT;

        IN_TASK.ORIGIN      := TASK_GUIDANCE  ;
        IN_TASK.DESTIN      := TASK_AUTO_PILOT;
        IN_TASK.ENT_CALL    := AP_ORDERS      ;

        INOUT.SEND (IN_TASK);
        accept ACK;
      or
        terminate;
      end select;
    end loop;

  end GUIDANCE;

  task EXE_MISSION is
    entry OBJECT_ALERT (EXE_MESSAGE : in MESSAGE_FORM);
```

```
    entry MONITOR_UPDATE (EXE_MESSAGE : in MESSAGE_FORM);
    entry ACK;
  end;

  task body EXE_MISSION is

-- Input is taken from MONITOR regularly and from AVOIDANCE in the case of
-- an obstacle problem (for the testing of this thesis, an obstacle
-- problem was assumed to always exist).  Output is always sent to
-- GUIDANCE.

    IN_TASK : MESSAGE_FORM;

  begin

    loop
      select
        accept MONITOR_UPDATE (EXE_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := EXE_MESSAGE;
        end MONITOR_UPDATE;

        accept OBJECT_ALERT (EXE_MESSAGE : in MESSAGE_FORM) do
          IN_TASK := EXE_MESSAGE;
        end OBJECT_ALERT;

        delay EXE_INT;

        IN_TASK.ORIGIN      := TASK_EXE_MISSION;
        IN_TASK.DESTIN      := TASK_GUIDANCE;
        IN_TASK.ENT_CALL    := UPDATE_ORDERS;
        INOUT.SEND (IN_TASK);

        accept ACK;
      or
        terminate;
      end select;
    end loop;

    accept MONITOR_UPDATE (EXE_MESSAGE : in MESSAGE_FORM) do
      IN_TASK := EXE_MESSAGE;
    end MONITOR_UPDATE;

  end EXE_MISSION;

  task MONITOR is
    entry TO_MONITOR (MON_MESSAGE : in MESSAGE_FORM);
    entry ACK;
```

```
    end;

    task body MONITOR is

    -- This dummy task simply gets input from VEHICLE_SYS and sends it on to
    -- EXE_MISSION.

      IN_TASK : MESSAGE_FORM;

    begin

      loop
        select
          accept TO_MONITOR (MON_MESSAGE : in MESSAGE_FORM) do
            IN_TASK := MON_MESSAGE;
          end TO_MONITOR;

          delay MONITOR_INT;

          IN_TASK.DESTIN     := TASK_EXE_MISSION;
          IN_TASK.ORIGIN     := TASK_MONITOR;
          IN_TASK.ENT_CALL  := MONITOR_UPDATE;

          INOUT.SEND (IN_TASK);
          accept ACK;
        or
          terminate;
        end select;
      end loop;

   end MONITOR;

begin

   loop
     MESSAGE_IO.READ (InFromMars, IN_MESSAGE);

     IN_MESSAGE.PROG(5) := IN_MESSAGE.PROG(5) + 1;

     if IN_MESSAGE.ORIGIN = SHUTDOWN then
       IN_MESSAGE.DESTIN    := HOST_TASK;
       IN_MESSAGE.PROG(5)  := -1 * IN_MESSAGE.PROG(5);
       INOUT.INCOMING (IN_MESSAGE);
       exit;
     end if;

     select
```

```
        INOUT.INCOMING (IN_MESSAGE);
      or
        delay SEND_INT;
        WAITING.LIMBO (IN_MESSAGE);
      end select;
    end loop;
  end PLUTO;
```

APPENDIX C: INVOKE AND LINKING FILES

A. MAKEFILE

This is the file used to create the necessary OCCAM libraries and generate all the files used in the harnesses. By simply invoking MAKE, all compilation and file generation was accomplished.

```
# File: makefile
#   "make help" to print option list
#
#   Complete development cycle:
#     make family       -- makes Ada family and library directories
#     make              -- compiles, links, configures source
#     make run          -- run bootable code

MODE = s
PROC = 8
OPTS = /$(MODE) /t$(PROC)

# make the executable code
main.btl: mainh.c$(PROC)$(MODE) marsh.c$(PROC)$(MODE) venush.c$(PROC)$(MODE)
saturnh.c$(PROC)$(MODE) plutoh.c$(PROC)$(MODE) main.pgm
   @ echo EXPECT 1 WARNING...Then cross your fingers and PRAY for the best!!
   iconf /s main.pgm
   @ f:\util\bell

mainh.c$(PROC)$(MODE): earth.o earthh.t$(PROC)$(MODE) merger.t$(PROC)$(MODE)
mainh.t$(PROC)$(MODE)
   ilink /f main.lnk

earth.o: common.ada printout.ada earth.ada
   ada invoke earth.inv,yes

earthh.t$(PROC)$(MODE): earthh2.tax earthh.occ
   occam $(OPTS) earthh.occ

earthh2.tax: earthh2.occ
   occam /ta /x earthh2.occ
```

```
merger.t$(PROC)$(MODE): merger.occ
    occam $(OPTS) merger.occ

mainh.t$(PROC)$(MODE): mainh.occ
    occam $(OPTS) mainh.occ

marsh.c$(PROC)$(MODE): mars.o marsh.t$(PROC)$(MODE)
    ilink marsh.t$(PROC)$(MODE) mars.o adarts8.lib hostio.lib occam8s.lib xlink.lib

mars.o: common.ada mars.ada
    ada invoke mars.inv,yes

marsh.t$(PROC)$(MODE): marsh2.tax marsh.occ
    occam $(OPTS) marsh.occ

marsh2.tax: marsh2.occ
    occam /ta /x marsh2.occ

venush.c$(PROC)$(MODE): venus.o venush.t$(PROC)$(MODE)
    ilink venush.t$(PROC)$(MODE) venus.o adarts8.lib hostio.lib occam8s.lib xlink.lib

venus.o: common.ada venus.ada
    ada invoke venus.inv,yes

venush.t$(PROC)$(MODE): venush2.tax venush.occ
    occam $(OPTS) venush.occ

venush2.tax: venush2.occ
    occam /ta /x venush2.occ

saturnh.c$(PROC)$(MODE): saturn.o saturnh.t$(PROC)$(MODE)
    ilink saturnh.t$(PROC)$(MODE) saturn.o adarts8.lib hostio.lib occam8s.lib xlink.lib

saturn.o: common.ada saturn.ada
    ada invoke saturn.inv,yes

saturnh.t$(PROC)$(MODE): saturnh2.tax saturnh.occ
    occam $(OPTS) saturnh.occ

saturnh2.tax: saturnh2.occ
    occam /ta /x saturnh2.occ

plutoh.c$(PROC)$(MODE): pluto.o plutoh.t$(PROC)$(MODE)
    ilink plutoh.t$(PROC)$(MODE) pluto.o adarts8.lib hostio.lib occam8s.lib xlink.lib

pluto.o: common.ada pluto.ada
    ada invoke pluto.inv,yes
```

```
plutoh.t$(PROC)$(MODE): plutoh2.tax plutoh.occ
    occam $(OPTS) plutoh.occ

plutoh2.tax: plutoh2.occ
    occam /ta /x plutoh2.occ

#
# misc.
#
help:
    @ echo Make arguments:
    @ echo   make              - make from top level down
    @ echo   make -n [opt]      - display but don't execute commands
    @ echo   make *.o           - make Ada object
    @ echo   make help          - display this list
    @ echo   make clean         - delete all files except source
    @ echo   make run           - run bootable program
    @ echo   make check         - check transputer topology
    @ echo   make family        - make Ada family and library directories

clean:
    del *.?8?
    del *.tax
    del *.o
    del *.dsc
    del *.btl
    del test_lib\adalib.*
    rd test_lib
    del test_fam\adafam.*
    rd test_fam

run:
    iserver /sb main.btl

check:
    check /r

family:
    ada invoke family.inv,yes
```

B. MAIN.PGM

The following OCCAM program is the program that assigns the compiled code to specific processors and sets up the hardware for the necessary link communication.

```
-- File: main.pgm

#INCLUDE "hostio.inc"
#INCLUDE "linkaddr.inc"

#USE "mainh.c8s"
#USE "marsh.c8s"
#USE "venush.c8s"
#USE "saturnh.c8s"
#USE "plutoh.c8s"

CHAN OF INT Mars2Earth, Earth2Mars, Venus2Mars, Saturn2Venus, Pluto2Saturn,
Mars2Pluto:
CHAN OF SP FromFiler, ToFiler:

PLACED PAR

  PROCESSOR 0 T8

    PLACE FromFiler  AT link0.in:
    PLACE ToFiler    AT link0.out:
    PLACE Mars2Earth AT link2.in:
    PLACE Earth2Mars AT link2.out:

    [325000] INT ws1:
    main.harness (FromFiler, ToFiler, Mars2Earth, Earth2Mars, ws1)

  PROCESSOR 1 T8

    PLACE Earth2Mars AT link0.in:
    PLACE Mars2Earth AT link0.out:
    PLACE Venus2Mars AT link2.in:
    PLACE Mars2Pluto AT link3.out:

    [280000] INT ws2:
    mars.harness (Mars2Earth, Earth2Mars, Venus2Mars, Mars2Pluto, ws2)

  PROCESSOR 2 T8

    PLACE Saturn2Venus AT link2.in:
    PLACE Venus2Mars   AT link3.out:

    [280000] INT ws2:
    venus.harness (Saturn2Venus, Venus2Mars, ws2)

  PROCESSOR 3 T8
```

```
    PLACE Pluto2Saturn AT link2.in:
    PLACE Saturn2Venus AT link3.out:

    [280000] INT ws2:
    saturn.harness (Pluto2Saturn, Saturn2Venus, ws2)

  PROCESSOR 4 T8

    PLACE Mars2Pluto   AT link2.in:
    PLACE Pluto2Saturn AT link3.out:

    [280000] INT ws2:
    pluto.harness (Mars2Pluto, Pluto2Saturn, ws2)
```

C. INVOKE FILES

These are simple files used by the MAKEFILE to generate the code.

```
-- File: family.inv
family.new test_fam,overwrite=yes
lib(family=test_fam).new test_lib,overwrite=yes

-- File: earth.inv
default.compile library=test_lib
compile common.ada
compile printout.ada
compile earth.ada
default.bind library=test_lib,level=bind,warning=no
bind earth,object="earth.o",entry_point="earth.program"

-- File: mars.inv
default.compile library=test_lib
compile common.ada
compile mars.ada
default.bind library=test_lib,level=bind,warning=no
bind mars,object="mars.o",entry_point="mars.program"

-- The remaining invoke files are identical except for the program names.
```

LIST OF REFERENCES

[ALSYS 90]
Alsys Inc., "PC Mothered Transputer Cross Compilation System User Manuals," Alsys, Burlington, MA, May 1990.

[CLOUTIER 90]
Cloutier, M. J., *Guidance and Control System for an Autonomous Vehicle,* Master's thesis, Naval Postgraduate School, Monterey, CA, June 1990.

[DATABOOK 89]
"The Transputer Databook," second edition, INMOS, Inc., Berkeley, CA, 1989.

[FLOYD 91]
Floyd C. A.,*Design and Implementation of a Collision Avoidance System for the NPS Autonomous Underwater Vehicle (AUV-II) Utilizing Ultrasonic Sensors*, Master's thesis, Naval Postgraduate School, Monterey, CA, September 1991.

[GESPAC 90]
GESPPU-1 Interface, GESPAC Press release, GESPAC, Inc., Geneva, SA, 1990.

[GOOD 89]
Good, M., *Design and Construction of the Second Generation AUV,* Master's thesis, Naval Postgraduate School, Monterey, CA, December 1989.

[G64 90]
Transputer Board Set Boosts Computing Capabilities of G-64 Systems, G64 Today #8, GESPAC, Inc., Geneva, SA, Fall 1990.

[HOARE 88]
Hoare C. A. R., series editor,"OCCAM 2 Reference Manual," INMOS, Inc., 1988.

[INMOS REF 86]
"Transputer Reference Manual", INMOS, Inc., Berkeley, CA,October 1986.

[MAKRIS 91]
Makris, D., *Real-time Scheduling and Synchronization for the NPS Autonomous Underwater Vehicle*, Master's thesis, NPS, Dec 1991.

[POUNTAIN 86]
Pountain D., "A Tutorial Introduction to OCCAM Programming," INMOS, Inc, March 1986.

[SKANSHOLM 89]
Skansholm J., "Ada From the Beginning",Addison Wesley Co., Menlo Park, CA, 1989.

[TRANS 89]
"Transputer Handbook", INMOS, Inc., Berkeley, CA, September 89.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Technical Information Center Cameron Station Alexandria, VA 22304-6145	2
2.	Library, Code 52 Naval Postgraduate School Monterey, CA 93943-5002	2
3.	Chairman, Code EC Department of Electrical & Computer Engineering Naval Postgraduate School Monterey, CA 93943-5002	1
4.	Professor Shridhar Shukla, Code EC/Sh Department of Electrical & Computer Engineering Naval Postgraduate School Monterey, CA 93943-5002	2
5.	Professor Roberto Cristi, Code EC/Cx Department of Electrical & Computer Engineering Naval Postgraduate School Monterey, CA 93943-5002	1
6.	Professor Robert B. McGhee, Code CS/Mz Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5002	1
7.	Professor Anthony Healey, Code ME/Hy Department of Mechanical Engineering Naval Postgraduate School Monterey, CA 93943-5002	1
8.	Professor Uno Kodres, Code CS/Kr Department of Computer Science Naval Postgraduate School Monterey, CA 93943-5002	1

DUDLEY KNOX LIBRARY
NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA 93943-5002

Thesis
R3946 Richmond
c.1 On programming transputers to capture Ada multitasking for the NPS autonomous underwater vehicle.

Thesis
R3946 Richmond
c.1 On programming transputers to capture Ada multitasking for the NPS autonomous underwater vehicle.

DUDLEY KNOX LIBRARY
3 2768 00018346 1