RSD-10-87

Distributed Run-Time Support for Ada on the NCUBE Hypercube Multiprocessor[1]

R.M. Clapp
T.N. Mudge
R. A. Volz

Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109

August 1987

Center for Research on Integrated Manufacturing

Robot Systems Division
College of Engineering
The University of Michigan
Ann Arbor, Michigan 48109-2110

[1]Ada is a registered trademark of the Department of Defense.

ensm

UMR1197

Abstract

This report discusses the design and implementation of a run-time system for Ada that is targeted to the NCUBE hypercube multiprocessor. A brief introduction to the hypercube architecture is given. The scope and structure of the run-time support is presented and its relationship to the target architecture is discussed. Actual implementation details are also included along with the prospects for expanding the support to a full distributed Ada run-time system.

Contents

1 Introduction

It is apparent from statements in the Ada Language Reference Manual (LRM) [LRM83] that Ada is not only intended for execution on shared memory multiprocessors but also on distributed processors that do not share memory. Accordingly, the LRM provides explanations of language constructs that consider the distributed program case. Unfortunately, the semantics are not clear for the distributed case. As a result, there have been several investigations into the issues and feasibility of distributing Ada. These discussions can be found in [VMB87], [VMN85], [Ard84] and [Cor84]. Related discussions of run-time support for distributed Ada can be found in [WEA84a], [WEA84b], [FiW86], [Ros87], and [Ard86]. In addition to the problems that arise at the language level when considering distribution, there are also issues concerning the target architecture. Particular consideration is given to this aspect of the problem in [VMB87].

The advent of inexpensive commercial hypercube multiprocessors [HMS86] has created an interest in targeting a distributed Ada compiler to these machines. This raises several issues, which will be discussed below. In order to provide some insight into the approach and feasibility of such a project, a partial run-time system for distributed Ada has been implemented on an NCUBE/ten, a hypercube multiprocessor configured for our work with 64 processors. The kernel implemented runs together with the existing operating system and includes support for executing Ada tasks in parallel. The distributed nature of the system comes from the assumption that the several tasks of an Ada program may reside on several processors (where the number of processors is equal to or less than the number of tasks). Support for memory management and I/O is included in the existing operating system.

This paper will provide a brief introduction to the hypercube architecture. Issues concerning this particular target architecture will also be discused. The structure of the system routines and their interface to user Ada programs will then be presented. Following that, implementation details concerning the target hardware and existing operating system are given. Finally, consideration is given to the expansion of the run-time system and to the feasibility of re-targeting an existing Ada compiler to the hypercube machine.

2 The Hypercube Architecture

A hypercube computer consists of several microprocessors interconnected by communication links. A hypercube computer of dimension n is a multiprocessor with $N = 2^n$ processors arranged at the nodes of a hypercube graph. Each processor is directly connected via communication links to n neighbors. An operating system kernel supports store and forwarding of messages so that processors that are not neighbors can still send messages to each other

through intermediate processors. The maximum number of distinct communication links that must be used to transfer a message between any two processors is n, the dimension of the hypercube.

In the case of the NCUBE machine, each processor has its own local memory (128 K-bytes) and the hypercube array is managed by a host computer (an Intel 80286 based system). The configuration used in our work at the University of Michigan is an NCUBE/ten with 64 processors. (More processors may be added to a total of 1024.) The processors are Vax-like 32-bit microprocessors with IEEE standard floating point capability. Each processor runs a copy of an operating system called Vertex that supports the store and forward communications noted above. The host runs a multi-user Unix-like operating system called Axis. Axis allows the hypercube array to be partitioned into subcubes that may be allocated to different users. More details on the NCUBE architecture can be found in [HMS86], while more information on Vertex can be found in [MBA86].

Although the host in this system can provide some degree of shared memory, for the most part, hypercube multiprocessors are distributed memory machines. This proved to be a source of some difficulty because the block structured nature of Ada permits shared variables, which translates to possible remote memory accesses. This issue and suggested solutions are discussed in detail in [VMB87]. References to remotely located variables and subprograms is beyond the scope of the initial run-time system discussed here. This report is concerned mainly with issues related to the distribution of tasks over several processors.

3 Overall System Structure

The partial run-time system that was implemented provides support for Ada tasking. More specifically, support is included for inter-task rendezvous, task delay, task elaboration, activation and termination, and task scheduling with time slicing. In all areas, the Ada rules governing parent-child task relationships are followed. As with the operating system, the run-time support kernel is replicated on each node in the hypercube that is being used.

Each node can run a collection of tasks. These tasks request service from the run-time system by making a call that generates an interrupt. The service is performed by the system and then another task is chosen to run on that node. The run-time system services requests by altering task states and data structures, and by sending messages. Processing of run-time system requests is carried out independently of whether any of the affected tasks are local or remote. This allows the tasks to be distributed across several processors. To support this, messages can be sent either locally or to other nodes. Incoming messages are read at scheduling points and placed on a queue along with all local messages. All pending messages are then processed once at each scheduling point. The basis for our implementation is a

message based system similar in philosophy to that proposed in [WEA84a] and [WEA84b].

3.1 Ada Tasking Control

The first major section of the kernel subset is tasking control. This part of the kernel contains routines for task elaboration and activation, task termination, task delay, and task rendezvous. Calls to these routines are made via interrupts by user tasks (i.e. Ada program tasks) when these operations need to be performed.

To provide a framework for the run-time system, we first define fourteen states that a task may be in. They are:

- Unborn
- Running in the declarative region
- Running in the body
- Running in a critical section
- Waiting to activate
- Delayed
- Waiting for a rendezvous accept message
- Waiting for a rendezvous return message
- Waiting for an entry call
- Waiting for an entry call confirmation message
- Completed
- Terminated
- Waiting to terminate
- Abnormal

Second, we currently define eleven different message types. Seven of these are associated with the rendezvous mechanism. The eleven types are:

- Simple entry call

- Conditional entry call
- Timed entry call
- Return from rendezvous
- Accept entry call
- Confirm entry call
- Abort entry call
- Elaborate child
- Child is active
- Child is completed
- Abort task

The use of the states and messages is explained in the sections below.

In addition to the task states and messages, there are also several data structures worth noting. Each task has a task control block that contains its name, parent's name, program counter, program status word, stack pointer, stack space, rendezvous information, and pointers to a state queue, child information, and entry information. A state queue is needed since some task states are temporary and the previous state must be remembered (e.g. a delayed task returns to its previous state when the delay expires). The child information is a linked list of nodes containing child names and their elaborated/active status. Entry information consists of the number of entries, the names and guards for each, the last entry serviced, and a message queue for each entry. Messages are also made up of several components depending on their type. Different fields include source, destination, identifier, and data. More details concerning the particular data structures are given as they arise in the discussion below.

3.1.1 Task Elaboration and Activation

There are two routines in the system to implement task elaboration and activation. These routines change the state of the requesting task and police the Ada rules concerning activation and parent-child task relationships. The elaboration of a task changes its state from unborn to running in the declarative region. The run-time system also records parent and entry point (for rendezvous) information of the newly elaborated task.

When a task calls the activation routine, its state changes to running in the body, unless it has children that are still not active. If the latter is true, the task enters a wait to activate state

and stays that way until an active message is received from all its children. An elaborate message is sent to all unborn children in this case to begin their elaboration. When the task's state changes to running in the body, a message is sent to its parent to enable the parent task to become active.

3.1.2 Task Completion and Termination

For these kernel operations, two routines are needed. One is for normal termination and the other is for abnormal termination. When a task is finished with its execution, the routine for normal termination is called. The requesting task becomes completed and, if all of its children are in the terminated state, it also becomes terminated. If any of its children are still active, it stays in the completed state and waits for a message from each child indicating that the child is terminated. Whenever a task enters the terminated state, it sends a message to its parent to allow the parent to terminate.

When a task wants another task to abort execution, a routine is called to perform the abnormal termination. The aborted task stops any rendezvous or delay in progress, enters the abnormal state and sends an abort message to all of its children. If the task has no children or its children are already terminated, the task enters the terminated state and sends a message indicating this to its parent.

3.1.3 Task Delay

There is one routine in the kernel to implement a task delay. The task requesting a delay passes the delay value to the routine. A linked list of task names and associated expiration times is kept by the run-time system. The task name-expiration pair is inserted into the list that is sorted by increasing expiration times. When the timer expires, the count of clock ticks since node startup is updated. This keeps track of the absolute time. A check is then made to see if any entries in the linked list need to be processed. The effective delay resolution then is roughly equal to the time slice value. The same linked list of task names and expiration times is used to implement time bounds placed on task rendezvous (discussed in next section).

3.1.4 Task Rendezvous

Three routines are needed to implement task rendezvous. One is needed for the calling task and two are needed for the accepting task. The calling task executes an entry call and the accepting task executes an accept statement.

3.1.4.1 Entry Call

There are three types of entry calls, the simple entry call, the conditional call, and the timed call. For a simple entry call, the calling task hangs until the entry call is accepted. A timed entry call requests the entry but withdraws its request if the rendezvous can not be started within the specified duration. If the time bound specified is zero, a conditional entry call is implemented where the rendezvous occurs only if the entry call can be accepted immediately. (Actually, there are problems with considering a zero delay to mean immediate. The check to determine if the entry can be accepted immediately takes time. Also, it may have been the intent of the task to not attempt the entry call based on a zero delay value. For more details on this issue and the interpretation of the Ada Language Maintenance Committee, see [VoM87]).

All forms of the entry call can be implemented in one routine. The calling task passes the name of the desired entry, the in parameters, the time bound, and a flag indicating if it is a timed entry call. If the flag is false and the time out value is greater than zero, a simple entry call is implemented. A time node is not created and a simple entry call message is sent. If the flag is false and the time bound is zero or negative, a conditional entry call results. If the entry is local, a check is made to see if the queue for that entry is empty and whether its guard is true. If this is the case, the entry call message is sent. If the entry is non-local, a conditional entry call message is sent to the proper node. This call is accepted only if the queue for the desired entry is empty and its guard is true. The final case is the timed entry call. Here the entry call message is sent and a time node is inserted into the linked list. If the call is not accepted by the timeout, an abort message is sent by the calling task to cancel the rendezvous. Whenever an entry call message is sent, the calling task enters the wait for accept state. All call and abort messages have identifiers associated with them so that only the proper calls are aborted.

3.1.4.2 Accept Statement

For the accepting task, there are several forms that can be used for selecting entries to be served. A simple accept statement hangs until the entry is called. A choice among several entries can be made using the select statement. Besides selecting among entries, the select may have a branch where code is executed if no entry is called within a specified duration. This branch may also be an else statement that is executed immediately if no entry calls are pending on any of the entries.

Since the accepting task executes the code during the rendezvous, two run-time system routines are needed to begin and end the task's critical section. Also, the first kernel routine called is needed to inform the task which branch of the select statement is to be executed (i.e. which entry was actually called).

The first routine, which begins the accept, is passed the accepting task's name, the values for the guards, and a timeout value. If the guard for the timer is true (i.e. the accept is part of a select with a delay branch), a time node is added to the linked list. A check is then made for each entry queue that has an open guard to see if an entry call is pending. If any entries are pending, the first one encountered is accepted. In the case of the timed entry call, an accept message is sent to the calling task and the accepting task waits for a confirmation message before executing the critical section. In the simple and conditional cases, the rendezvous begins immediately with a message being sent to the caller upon completion of the critical section. The search for a pending entry begins one past the last entry accepted to ensure fair service to all entries. If no entries are pending, the accepting task enters a wait for entry call state. In this case, the task will hang until an entry call is made to it or the timeout expires (if present). A guard value of true is assigned to an else branch of a select statement if it is present. If no entry calls are pending in this case, a fictitious call to the else is accepted.

The second routine is called after successful completion of a rendezvous by the accepting task. At this point, the critical section is exited and a return message is sent to the calling task to allow it to proceed with its execution. This routine is passed the name of the accepting task and the value of the out parameters of the rendezvous.

In all three routines supporting task rendezvous, certain information must be passed to the tasks requesting service. Since calls are generated by interrupts and conventional returns are not used, information is passed back to a task in space reserved in its task control block. Such information includes a flag to indicate successful rendezvous, index into a select of the entry accepted, and values for in and out parameters.

3.1.4.3 Rendezvous Message Passing

These three routines for task rendezvous perform the necessary synchronization through altering task states and sending messages. A timed entry call results in an call message being sent to the desired entry. When the accepting task is ready to accept that entry (or if it already is ready) an accept message is sent to the calling task. If the calling task has not timed out on the call, a confirm message is sent to the entry to start the rendezvous. When the confirm message is received, the critical section is executed and the rendezvous is completed. A return message is then sent to the caller so that it may resume execution. If the calling task times out before receiving an accept message, an abort message is sent to cancel the rendezvous. Since the calling task may be in a loop and again call the entry before the abort is received, identifiers are combined with task name for each rendezvous related message so that they may all be uniquely identified.

In the case of a simple entry call, the number of messages passed is reduced to two. Since there is no possibility of a time out, once an entry call message is sent, the calling

task waits for a return message from the called task. Thus, there are no accept or confirm messages sent.

The message passing for conditional entry calls is also simplified. For local calls, a check is made on the desired entry to see if a call to it can be accepted. If so, a simple entry call to that entry is performed. If the call is to a remote entry, a conditional entry call message is sent. When it is received, a check is made to see if it can be serviced immediately. If it can, the call is processed and a reply is sent upon completion of the rendezvous. If it can not be immediately accepted, a negative reply is sent to the caller.

3.2 Communications and Control

The other major section of the kernel is communications and control. The routines here are not visible to the user's tasks. They provide the message sending and receiving operations, task scheduling, and timing management. This part of the kernel also must handle the target system interface.

3.2.1 Message Passing

Two routines are support this operation, one for sending messages and one for receiving them. The message sending routine determines whether the message destination is on a local or remote node. If the message is local, it is placed on a message queue and will be processed at the next scheduling point. Non-local messages are sent to the destination node via the operating system communications support.

The message receiving routine dequeues messages and performs any necessary operations. A message usually requires a state change for a task, an outbound message to be sent, or some other update of a task's data structures. When messages from remote nodes are received, they are placed on the local message queue for processing along with all other messages. At each scheduling point, before a new task is selected to run, all pending messages are processed.

3.2.2 Timing Support

Two routines are needed here to set time nodes in the linked list (mentioned earlier) and process a timeout condition. These routines include code to update the linked list of expiration times and code to make state changes and send messages. Also necessary here is a modification to the operating system interrupt handler for a timeout interrupt. The handler reloads the timer with the time slice value and adds this value to a counter of clock ticks.

The routine that processes timeout conditions is also called from within the timeout interrupt handler.

Since tasks yield control of the CPU whenever a call to the run-time or operating system is made, it is possible that they will not use their entire time slice. In this case, before the next task is dispatched, the timeout register is read and the number of ticks since the last time slice was loaded is computed and added to the counter of clock ticks. This updates the absolute time since node startup. The timeout register is then loaded with the time slice value for the next task. Whenever a value for the current time is needed (e.g. when adding a node to the linked list of timeouts), a function is called that multiplies the counter of clock ticks by a system constant to obtain the absolute time since startup in seconds.

The complexity of the time management strategies is due to the fact that there is only one timer on each node. This scheme also prevents clock synchronization across nodes and allows for the time value to drift if interrupts are masked during manipulation of timer dependent data structures. It would be far more desirable to have a real time clock register on each node as well as the interval timer currently provided. This would allow an accurate value for the current time to always be available on all nodes and would also reduce the overhead in processing interrupts generated by the timeout register.

3.2.3 Task Scheduler

At the high level, the task scheduler makes a call to receive all pending messages and then picks a task to execute. The scheduling of tasks is round robin among executable tasks. All task pointers are included in a linked list. Terminated tasks are removed from the list when they are encountered. The list is traversed until an executable task is found or a complete cycle is made. If a cycle is made and no task is executable, the message queue is checked to see if any new messages need to be read. If not, the timer linked list is checked to see if any timeouts will occur in the future. If the timer list is null and no messages are pending, deadlock occurs for the single node case. In the multiple node case, there still may be messages coming from other nodes. When the linked list of task pointers becomes null, all tasks are terminated and the host program is informed by the scheduler that the node has completed its execution. When all nodes allocated complete execution, the host program terminates.

4 Implementation Details for the NCUBE Machine

4.1 Approach

The code for this run-time system was written in C, a language with a compiler for it on the NCUBE. A simulation version was first implemented in C on a Vax 11/780. This step was performed to reduce debug time necessary on the NCUBE, where the debugging tools are more primitive.

4.1.1 Operating System Interface

In order to run the system in C on the NCUBE, the operating system running on the nodes had to be modified. Assembly language routines were written to generate run-time system calls. These calls are implemented via interrupts. The interrupt vector table provides the interrupt handler's address and a new program status word. The new program status word in all cases enables interrupts so that communications between the nodes can continue. This does not interfere with the operation of the run-time system. Timer interrupts are also allowed, so that the count of clock ticks can be maintained accurately. Processing of the timer interrupt includes examining the program counter of the interrupted code to determine if a user task or the run-time and operating system was interrupted. The next step for the interrupt handler of the called run-time system routine is to move the parameters from the calling task's stack to the run-time system stack area. All registers are then saved in the calling task's stack area contained within its own task control block. The stack pointer is changed to point to the system stack area and the call to the appropriate run-time system routine is made. Upon its return, a call is made to the task scheduler which also makes a call to receive pending messages. Upon return, the registers of the selected task are reloaded, the stack pointer changed, and the task given control via a return from interrupt instruction that restores the program counter and program status word. The same procedure occurs for a timeout interrupt of a user task, except that maintenance of the time variables and processing of timeouts is also performed. Direct calls to Vertex from user tasks also result in a context switch and scheduling of a new task. All tasks picked to run by the scheduler are given a fresh time slice before being dispatched. This requires updating of the counter of clock ticks as mentioned earlier.

The run-time system routines are written in C and the operating system (Vertex) is written in assembly language. Since they are not linked together and Vertex is entered through interrupts, addresses of run-time system variables and routines must be passed to Vertex. An initialization interrupt handler is used to load a table of the necessary addresses. This allows both Vertex and the run-time system to share global data.

4.1.2 Form of User Program

Currently, Ada programs are hand translated into sequential C code and run-time system calls. Each task then appears as a separate C routine. The main unit of this program is a routine that is executed once at startup to perform data structure allocation and initialization. Task control blocks are allocated from high memory using getbuf in Vertex. (This was done to keep stack pointers out of the user space, because this condition confuses the C compiler and generates an error). Other space is allocated from the C routine malloc for task state queues, child information, and entry data. The program counter, program status word, and stack pointer are initialized for each task. Finally, this routine generates the initialize system interrupt to pass important system addresses to Vertex. After the table is initialized, control is passed to the task that represents the main unit of the Ada program.

4.2 Example of Run-Time System Interface

To demonstrate the relationship between an Ada program and the underlying run-time system, an example is given in the Appendix. The original user code is given as an Ada program. Static specification of the location of tasks is done through the use of a SITE **pragma** as proposed in [VMN85]. The resulting sequential code and run-time system calls are given in C. This is the code that is given to the C compiler and linked with the run-time system. Each C routine given represents a separate task and both tasks in this example reside on a unique processor in the network. The two C programs are compiled separately and each is linked to its own copy of the run-time system (i.e. each node in use has its own copy of the run-time system as well as Vertex).

5 Future Directions

At this point, a partial distributed run-time system for Ada on the NCUBE machine has been implemented. However, there are still some features that should be part of this run-time system subset that have yet to be implemented. These features include support for dynamic task creation and the **terminate** alternative in a **select** statement. Implementation of these features, though, is greatly dependent on the distribution strategies adopted. This issue is investigated in [VMB87], but must be re-examined in the case of the hypercube architecture. Then, the above mentioned language features may be incorporated into the run-time system as well as additional components (e.g. exception handling, remote procedure call, remote object reference, etc.).

A great advantage of the existing system at this point is the fact that it is coded in a high

level language (C) and interfaced with the low level operating system (Vertex). There is also existing support for memory allocation and I/O in Vertex. Altogether, this allows the content of the run-time system to be easily modified and re-compiled. A distributed run-time system testbed has been established for the NCUBE machine, allowing for further experimentation to proceed smoothly.

The run-time system can now be extended to include full run-time support for distributed Ada or even distributed Concurrent C [CGP85]. Since the user tasks are now in the form of C routines, targeting a Concurrent C compiler to the NCUBE to interface with this run-time system should be an easier task. However, in either case, the step of extending the run-time system to be complete requires an examination of the issues involved with each new component included. An attempt at implementation, though, can uncover the issues involved. With the testbed established, prototyping of the expanded run-time system can proceed rather quickly. At any rate, in the case of Ada or Concurrent C, successful completion of a complete run-time system would allow the re-targeting of a compiler to the NCUBE.

References

[Ard84] Ardö, A., "Experimental Implementation of an Ada Tasking Run-Time System on the Multiprocessor Computer Cm*," *Proceedings of the 1st Annual Washington Ada Symposium*, pp. 145-153, March 1984.

[Ard86] Ardö, A., "Efficiency Aspects on Ada Run-Time Support for Multiprocessors with Shared Memory," Technical Report, Department of Computer Engineering, University of Lund, March 1986.

[CGP85] Cmelik, R.F., N.H. Gehani, M. Plotnick and W.D. Roome, "Concurrent C Project," AT&T Bell Laboratories, 1985.

[Cor84] Cornhill, D. "Partitioning Ada Programs for Execution on Distributed Systems," *1984 Computer Data Engineering Conference*, 1984.

[FiW86] Fisher, D.A. and R.M. Weatherly, "Issues in the Design of a Distributed Operating System for Ada," *IEEE Computer*, pp. 38-47, May 1986.

[HMS86] Hayes, J.P., T.N. Mudge, Q. Stout, S. Colley and J. Palmer, "A Microprocessor-based Hypercube Supercomputer," *IEEE Micro*, pp. 6-17, October 1986.

[LRM83] *Ada Programming Language (ANSI-MIL-STD-1815A)*. Washington, D.C. 20301: Ada Joint Program Office, Department of Defense, OUSD (R&D), January 1983.

[MBA86] Mudge, T.N., G.D. Buzzard and T.S. Abdel-Rahman, "A High Performance Operating System for the NCUBE," *Proceedings of the 1986 SIAM Conference on Hypercube Multiprocessors*, pp. 90-99, September 1986.

[Ros87] Rosenblum, D.S., "An Efficient Communication Kernel for Distributed Ada Run-Time Tasking Supervisors," *ACM Ada Letters*, vol. 7, no. 2, pp. 102-117, March/April 1987.

[VMB87] Volz, R.A., T.N. Mudge, G.D. Buzzard and P. Krishnan, "Translation and Execution of Distributed Ada Programs: Is It Still Ada," *IEEE Transactions on Software Engineering*, (to appear).

[VMN85] Volz, R.A., T.N. Mudge, A.W. Naylor and J.H. Mayer, "Some Problems in Distributing Real-Time Ada Programs Across Machines," *Ada in use, Proceedings of the 1985 International Ada Conference*, Paris, France, May 1985, pp. 72-84.

[VoM87] Volz, R.A. and T.N. Mudge, "Timing Issues in the Distributed Execution of Ada Programs," *IEEE Transactions on Computers*, vol. c-36, no. 4, pp. 449-459, April 1987.

[WEA84a] Weatherly, R.M. "Design of a Distributed Operating System for Ada," Ph.D. Thesis, Clemson University, August 1984.

[WEA84b] Weatherly, R.M. "A Message-Based Kernel to Support Ada Tasking," *IEEE Computer Society 1984 Conference on Ada Applications and Environments*, pp. 145-152, October 1984.

Appendix

The following example shows how an Ada program with tasking is represented as C code with run-time system calls.

A Original Ada Program

```
with TEXT_IO; use TEXT_IO;
procedure MAIN is
--  This program consists of a tasks and a main program (second task).
--  The main program executes an entry call to the first task and passes
--  it a value.  It then executes a timed entry call to the task to read
--  that value. It then checks to confirm that the value it originally
--  passed to the task is the same as the value read from the task.
--  The main program then delays, aborts the task (since it is in an
--  infinite loop), and completes.

   PARM_VAL : INTEGER := 1;
   READ_VAL : INTEGER;

   task FIRST_LINK is
      entry DEPOSIT(VAL : in  INTEGER);
      entry READ    (VAL : out INTEGER);
   end;

   task body FIRST_LINK is
      LOCAL_VAL : INTEGER;
   begin
      loop
         accept DEPOSIT(VAL : in INTEGER) do
            LOCAL_VAL := VAL;
         end;
         accept READ(VAL : out INTEGER) do
            VAL := LOCAL_VAL;
         end;
      end loop;
   end FIRST_LINK;

   pragma SITE(0, MAIN);
   pragma SITE(1, FIRST_LINK);

begin      /* Body of procedure Main */

   FIRST_LINK.DEPOSIT(PARM_VAL);     --Entry call to pass value to first task
```

```
   select
      FIRST_LINK.READ(READ_VAL);
      if (PARM_VAL = READ_VAL) then
         PUT_LINE("Value passed was unchanged.");
      else
         PUT_LINE("Value passed was changed.");
      end if;
   or
      delay 10.0;
      PUT_LINE("Value couldn't be read from task within 10 seconds.");
   end select;

   delay 2.0;
   abort FIRST_LINK;

end MAIN;
}
```

B C Code With Run-Time System Calls

B.1 C Program for Unit MAIN

```
#include "types_spec.h"          /*Include file with type definitions.*/
extern TIME_PNT   time_root;          /*Root of linked list of times.*/
extern BLOCK_PNT  ctask;              /*Pointer to tcb of current task.*/
extern   int      msg_count;          /*Number of pending messages.*/
extern MSG_PNT    msg_store[max_msg]; /*Queue of messages.*/
extern   int      last_msg;           /*Last message read in queue.*/
extern   int      time_count;         /*Number of clock ticks since startup.*/
extern   int      t_slice;            /*Time slice value.*/

/* The following routines cause interrupts so Vertex can save the state
   of the executing task before performing the specified service in the
   run-time system.                                                      */
extern elaborate_m();    /*Task elaboration routine. */
extern activate_m();     /*Task activation routine. */
extern terminate_m();    /*Task termination routine. */
extern delay_m();        /*Task delay routine. */
extern accept_begin_m();/*Routine to begin accept of rendezvous.*/
extern accept_end_m();   /*Routine to end critical section of rendezvous.*/
extern entry_m();        /*Routine to execute entry call. */
extern abort_m();        /*Task abort routine. */
extern child_m();        /*Routine to establish parent-child relationship.*/
extern init_sys();       /*Routine to initialize system and pass addresses
```

```
                                        to Vertex. */

/* This file contains all code for each user task.  Conceptually, this
   can be thought of as the code generated by an Ada compiler to be
   further input to a C compiler and linked to the run-time system.    */

t_0_code()    /* Code for procedure Main */
{
        char logstr[132];     /* String for NCUBE I/O */

        PARAMETER_LIST          parms_1;
        PARAMETER_LIST          parms_2  = NULL;
        int                     read_val;
        int                     parm_val  = 1;

   parms_1 = malloc(sizeof(PARM));
   parms_1->next_value = NULL;

   elaborate_m(NULL_TASK, NULL_ENTRY);  /* the 0 is the name of task 0,
                                           main program has no parent and no entries.*/
   activate_m();                       /* the 0 is the name of task 0 */
   parms_1->value = parm_val;

   /* Entry call from task 0 to entry 1(FIRST_LINK.DEPOSIT),
                                      condition to set timer is FALSE. */
   entry_m(1, parms_1, INFINITY, FALSE);

   /* Timed entry call from task 0 to entry 2(FIRST_LINK.READ),
                                     condition to set timer is TRUE.  */
   entry_m(2, parms_2, 10.0, TRUE);
   if (ctask->accepted) {
      read_val = ctask->out_parms->value;
      if (parm_val == read_val) {
         sprintf(logstr, "Value passed was unchanged.\n");
         syslog(777,logstr);
      }
      else {
         sprintf(logstr, "Value passed was changed.\n");
         syslog(777,logstr);
      }
   }
   else {
      sprintf(logstr,
              "Value could not be read from last task within 10 seconds.\n");
      syslog(777,logstr);
   }
   delay_m(2.0);  /* Delay for 2 seconds. */
   abort_m(1);   /* Abort task 1 (FIRST_LINK) */
```

```
   terminate_m(); /* Terminate main program. */
}   /* End of Ada procedure Main */

t_1_code()    /* Last task is null to generate name for highest address in
 user task code space.    */
{
}

#define NUM_TASKS       1    /* One task on this site. */

static  BLOCK_PNT       tasks[NUM_TASKS];

main ()   /* C routine to initialize program running on this node. */
{
        int             i;
        CHILD_PNT       point;
        char            logstr[132];


/*  Loop to initialize task control blocks.  The call to mmalloc calls my
    assembly routine my_malloc.  This allocates memory from the message
    buffer area.  This is done to prevent the system routine check stack
    from incorrectly signaling a stack overflow.                          */

   for (i = 0; i <= (NUM_TASKS - 1); i++) {

      tasks[i]                = (BLOCK_PNT)mmalloc(sizeof(CONTROL_BLOCK));
      tasks[i]->psw           = 0x1f0000;
      tasks[i]->sp            = tasks[i];
      tasks[i]->sp            = tasks[i]->sp + (sizeof(CONTROL_BLOCK) - 64);

      tasks[i]->name                    = i;
      tasks[i]->state_queue             = (STATE_PNT)malloc(sizeof(STATE_NODE));
      tasks[i]->state_queue->state      = UNBORN;
      tasks[i]->state_queue->next_state = NULL;
      tasks[i]->state_queue->caller     = NULL_TASK;
      tasks[i]->state_queue->acceptor   = NULL_TASK;
      tasks[i]->dependents              = NULL;
      tasks[i]->id_generator            = 0;
   }

/* Set up a linked list of tcb's. */

   for (i = 0; i <= (NUM_TASKS - 1); i++)
      tasks[i]->next_block = tasks[(i + 1)];
   tasks[(NUM_TASKS - 1)]->next_block = tasks[0];

/* Initialize parent child relationships through a linked list of dependent
```

```
   information.                                                    */

       tasks[0]->dependents              = (CHILD_PNT)malloc(sizeof(CHILD));
       tasks[0]->dependents->name        = 1;
       tasks[0]->dependents->elaborated = FALSE;
       tasks[0]->dependents->active      = FALSE;
       tasks[0]->dependents->next_child = NULL;

   /* Task 0 starts from the init handler and does not have all of its
      registers popped from the stack before its initial execution.
      Therefore it does not need the extra 60 bytes subtracted from
      its stack pointer like the other tasks do.  See vertex listing. */

   tasks[0]->sp          = tasks[0]->sp + 60;

   /* Initialize program counter. */

   tasks[0]->pc          = t_0_code;

/* Allow task for main unit to be scheduled by making it runnable */
   tasks[0]->state_queue->state = RUN_DCL;

   ctask                 = tasks[0];
   last_msg              = -1;
   msg_count             = -1;

/* Pass routine and ctask addresses to vertex. */

   init_sys(&ctask, &time_count, &t_slice, entry_mon, accept_begin_mon,
            accept_end_mon, delay_mon, child_mon, elaborate_mon,
            activate_mon, abort_mon, terminate_mon, answer_timer,
            scheduler, t_0_code, t_1_code);
}
```

B.2 C Program for Unit FIRST_LINK

```
/* This program also contains the external declarations found above
   in the program for the unit MAIN.  They are omitted here to save
   space and avoid duplication.                    */

t_0_code()          /* Code for task FIRST_LINK */
{
        char logstr[132];

        BOOLEAN         local_guards[3];
        int             i;
```

```
        PARAMETER_LIST  parms_1 = NULL;
        PARAMETER_LIST  parms_2;
        ENTRY_PNT       entry_data;
        int             local_val;

/* Allocate and initialize space for parameter list and entry data. */
              parms_2 = malloc(sizeof(PARM));
              parms_2->next_value = NULL;
              entry_data = (ENTRY_PNT)malloc(sizeof(ENTRY_RECORD));
              entry_data->last_entry = 0;
              entry_data->entry_size = 2;
              entry_data->entries   = malloc(3 * sizeof(int));
              entry_data->guards    = malloc(3 * sizeof(int));
              entry_data->msg_queue = malloc(3 * sizeof(MSG_NODE_PNT));
              for (i = 0; i <= 2; i++) {
          entry_data->entries[i] = i;       /* i is name of entry */
          entry_data->guards[i] = FALSE;
          entry_data->msg_queue[i]  = NULL;
              }
              entry_data->entries[0] = 0;

elaborate_m(0, entry_data);  /* Elaborate task 1 with parent 0 and entries.*/
activate_m();                  /* Activate task 1 */
while (TRUE) {                  /* loop */
   local_guards[0] = FALSE;  /* Guard for delay part or else part. */
   local_guards[1] = TRUE;   /* Want to accept first entry. */
   local_guards[2] = FALSE;
   accept_begin_m(local_guards, 0.0);  /* 0 is time out value (not used). */
   switch(ctask->index) {
      case 1:
       local_val = ctask->in_parms->value;
       accept_end_m(parms_1);
       break;
      default:
       sprintf(logstr, "Error, simple accept must hang until entered.\n");
       syslog(777,logstr);
       break;
   }
   local_guards[1] = FALSE;
   local_guards[2] = TRUE;    /* Accept second entry */
   accept_begin_m(local_guards, 0.0);
   switch(ctask->index) {
      case 2:
       parms_2->value = local_value;
       accept_end_m(parms_2);
       break;
      default:
       sprintf(logstr, "Error, simple accept must hang until entered.\n");
```

```
        syslog(777,logstr);
        break;
    }
}                   /* end loop; */
terminate_m();
}

t_1_code()     /* See above comment for null routine purpose. */
{
}

#define NUM_TASKS       1     /* One task on this site. */

static  BLOCK_PNT       tasks[NUM_TASKS];

main ()
{
        int             i;
        CHILD_PNT       point;
        char            logstr[132];

/*  Loop to initialize task control blocks.  The call to mmalloc calls my
    assembly routine my_malloc.  This allocates memory from the message
    buffer area.  This is done to prevent the system routine check stack
    from incorrectly signaling a stack overflow.                          */

   for (i = 0; i <= (NUM_TASKS - 1); i++) {

      tasks[i]                = (BLOCK_PNT)mmalloc(sizeof(CONTROL_BLOCK));
      tasks[i]->psw           = 0x1f0000;
      tasks[i]->sp            = tasks[i];
      tasks[i]->sp            = tasks[i]->sp + (sizeof(CONTROL_BLOCK) - 64);

      tasks[i]->name                    = i + 1;
      tasks[i]->state_queue             = (STATE_PNT)malloc(sizeof(STATE_NODE));
      tasks[i]->state_queue->state      = UNBORN;
      tasks[i]->state_queue->next_state = NULL;
      tasks[i]->state_queue->caller     = NULL_TASK;
      tasks[i]->state_queue->acceptor   = NULL_TASK;
      tasks[i]->id_generator            = 0;
   }

/* Set up a linked list of tcb's. */

   for (i = 0; i <= (NUM_TASKS - 1); i++)
      tasks[i]->next_block = tasks[(i + 1)];
   tasks[(NUM_TASKS - 1)]->next_block = tasks[0];
```

```
/* Initialize parent child relationships through a linked list of dependent
   information.                                                          */
       tasks[0]->dependents              = NULL;

   /* Task 0 starts from the init handler and does not have all of its
      registers popped from the stack before its initial execution.
      Therefore it does not need the extra 60 bytes subtracted from
      its stack pointer like the other tasks do.  See vertex listing. */
   tasks[0]->sp              = tasks[0]->sp + 60;

   /* Initialize program counter. */
   tasks[0]->pc              = t_0_code;

   ctask                     = tasks[0];
   last_msg                  = -1;
   msg_count                 = -1;

/* Pass routine and ctask addresses to vertex. */
   init_sys(&ctask, &time_count, &t_slice, entry_mon, accept_begin_mon,
            accept_end_mon, delay_mon, child_mon, elaborate_mon,
            activate_mon, abort_mon, terminate_mon, answer_timer,
            scheduler, t_0_code, t_1_code);
}
```