

AD-A132 745

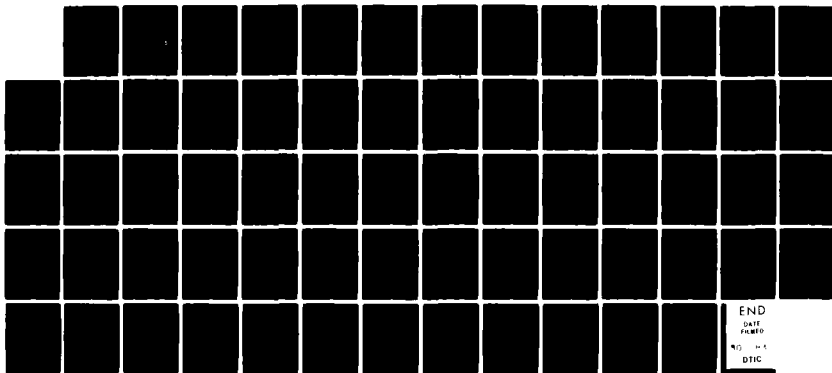
IMPLEMENTING AN ADA KERNEL ON NEBULA(U) TEMPLE UNIV
PHILADELPHIA PA G P INGARGIOLA AUG 83 ARO-18023.1-EL-R
DAAG29-81-K-0059

1/1

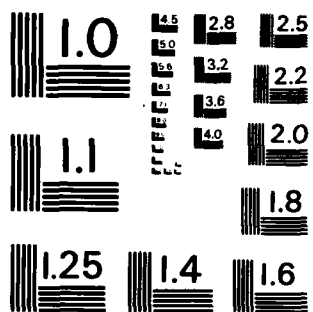
UNCLASSIFIED

F/G 9/2

NL



END
DATE
FILMED
DTIC



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

**READ INSTRUCTIONS
BEFORE COMPLETING FORM**

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report reviews the concurrency features of ADA, examines the aspects of the NEBULA architecture that are more significant for the implementation of concurrent programs, suggests a method for reducing the tasking mechanisms of ADA to a few simple kernel operations, and evaluates the NEBULA architecture in terms of this method and these operations.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ADA 132 745

BEST AVAILABLE COPY

IMPLEMENTING AN ADA KERNEL ON NEBULA

Giorgio P. Ingargiola
Temple University

FINAL REPORT(*)

U. S. ARMY RESEARCH OFFICE
CONTRACT DAAG29-81-K-0059

APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED



Accession For	
NTIS GRA&I	☒
DTIC TAB	☐
Unannounced	☐
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A	

(*) The views, opinions, and/or findings contained in this report are those of the author and should not be construed as an official Department of the Army position, policy, or decision, unless so designated by other documentation.

83 09 20 197

FOREWORD

NEBULA [NEB] and Ada [ADA] are rapidly approaching use. ^{These} ~~They~~ have been designed to achieve similar goals, to improve the quality, timeliness, and cost of the real-time systems used in the Department of Defense.

NEBULA is intended to provide a common architecture for these systems in applications that range from microprocessors to mainframes. Ada is designed, among other things, to ease concurrent programming.

This report

- reviews the concurrency features of Ada,

- examines the aspects of the NEBULA architecture that are more significant for the implementation of concurrent programs,

- suggests a method for reducing the tasking mechanisms of Ada to a few simple kernel operations, and

- evaluates the NEBULA architecture in terms of this method and these operations .

It is found that NEBULA supports admirably the control structures of Ada, but its Memory Management system is not very suitable. Entry calls and Accept statements are found to require about 40 machine instructions each. A simple rendez-vous requires the execution of over 100 instructions.



TABLE OF CONTENTS

Foreword	i
Table of Contents	ii
List of Illustrations	iii
1.0 Introduction	1
2.0 Concurrency in Ada	3
2.1 The Storage Structure of Ada Programs	7
2.2 Sharing Data across Tasks	15
2.3 More on Concurrency	16
3.0 Overview of NEBULA	18
3.1.0 The Address Spaces of NEBULA	18
3.1.1 The Physical Address Space	18
3.1.2 The Virtual Address Space	18
3.1.3 Virtual to Physical Address Translation	19
3.1.4 Memory Management Traps	21
3.1.5 Hardware Support for Replacement Policy	21
3.1.6 Setting up and Modifying Virtual Address Spaces	21
3.1.7 Some special Memory Management Instructions	22
3.2 Procedures in NEBULA	22
3.3 Context Stacks	24
3.4 The Processor Status Word	25
3.5 Controlling the Composite State of a Program	26
3.6 Task Oriented Instructions	27
3.7 The NEBULA model of Tasks and of the Tasking Kernel	28
4.0 A Concurrency Kernel for Ada	35
4.1 Task Control Blocks	35
4.2 Singly Linked Queues and Semaphores	39
4.3 Some basic Task Interactions	44
4.4 Implementing the Ada Kernel on NEBULA	56
Summary	59
Bibliography	61

LIST OF ILLUSTRATIONS

- Figure 2.1: A Block Structured Program
- Figure 2.2: Activation Frames
- Figure 2.3: Activation Frames on NEBULA
- Figure 2.4: Relation between stacks of Master and Dependent tasks
- Figure 3.1: The Virtual Address Space of NEBULA
- Figure 3.2: The Physical Address Space of NEBULA
- Figure 3.3: Memory Management Maps and Registers
- Figure 3.4: SVC Table
- Figure 4.1: Template of Task Control Block
- Figure 4.2: The Compare and Swap operation
- Figure 4.3: Specification of the QUEUE package
- Figure 4.4: Body of the QUEUE package
- Figure 4.5: Specification of the SEMAPHORE type
- Figure 4.6: Implementation of the SEMAPHORE type
- Figure 4.7: Some Assumed Program Components
- Figure 4.8: Code for Simple Entry Call
- Figure 4.9: Code for Simple Accept Statement
- Figure 4.10: Code for Conditional Entry Call
- Figure 4.11: Code for Timed Entry Call
- Figure 4.12: Code for Selective Wait Statement

1.0 INTRODUCTION

In this report we examine the problem of implementing the concurrent aspects of Ada* in terms of more simple mechanisms and of implementing these mechanisms in terms of the NEBULA Instruction Set Architecture. Our aim is to determine the suitability of Ada for implementing the real-time systems that will execute on NEBULA machines.

The NEBULA Architecture has been introduced to provide an advanced common Instruction Set Architecture for real-time systems used by the Department of Defense (DOD). This common architecture is assumed to be shared by machines with widely divergent performance characteristics, from Microcomputers, to Minicomputers, to Mainframes. It is intended to permit object-level shareability of code among all members of this computer family.

NEBULA is an efficient [DS], modern 'computer' that has features and instructions especially designed to support concurrent programming. Tasks are well defined objects of this architecture with operations for starting a new task and for performing a context switch. It can communicate with I/O Controllers by messages.

The Ada Language was developed with the intent of making it the standard implementation languages for the real-time systems used by the DOD. It is a modern strongly-typed Language with features that:

1. Support Modularity with Packages, Separate Compilations, controlled name spaces.
2. Improve Portability with the use of Pragmas, Representation statements, the predefined System Package, and with standard mechanisms for interfacing to other languages and to machine code.
3. Permit the direct expression of concurrency by allowing task definitions and permit intercommunication among tasks by using the convenient rendez-vous mechanism.
4. handle with the Exception Handling mechanism the errors and the abnormal contingencies that are so common in real-time applications

Normally, real-time systems are written in a mixture of high-level language, assembly language, and a heavy dose of calls to services of the underlying operating system. With the advent of Ada, in theory at least, the whole program can be written in Ada in a manner that is independent of the computer and of the operating systems being used (and when dependencies exist, they can be carefully isolated).

* Ada is a registered trademark of the U.S. Government, Ada Joint Program Office

In this report we examine the question of how well NEBULA supports the concurrency features of Ada. We address this question by examining how the rendez-vous mechanism of Ada can be expressed in terms of more primitive operations and how well these operations can be implemented in NEBULA. Our approach is essentially by example. No comparative study to other architectures in the style of [TAN] is attempted.

This report starts by examining the Tasking Mechanism of Ada. Particular consideration is given to the storage structure of Concurrent Programs because tasks in Ada are assumed to have direct access to the objects visible to them.

The features of the NEBULA Architecture that more directly impact the implementation of a concurrency kernel are next examined. Questions of Memory Spaces and of Memory Management are addressed. The Procedure call mechanism, central to the NEBULA architecture, is examined.

Control Blocks are specified for Ada Tasks. Simple operations are defined for implementing the various forms of the rendez-vous mechanism.

Some of the concurrent features of Ada are implemented in terms of sequential Ada and of these simple operations. The techniques used can be directly extended to cover all of concurrent Ada. This study is independent of the particular features provided by NEBULA.

Finally it is discussed how the concurrency kernel of Ada can be implemented in NEBULA. The aim here is to achieve a good understanding of the suitability of NEBULA for this implementation. Where determined, the number of NEBULA instructions required to carry out particular operations are given. Since NEBULA has very powerful instructions that are likely to be time consuming, the figures that are given may be misleading. Partial conclusions on the suitability of the Ada/NEBULA combination for implementing real-time systems are offered.

2.0 CONCURRENCY IN ADA

The specification part of an Ada task describes the entries that the task makes available to its environment. Entries are distinguished into Simple Entries and Entry Families.

Simple entries have forms like:

```
entry RESERVE (A: in TRACK_INDEX);      -- TRACK_INDEX is, say, an
                                         -- enumeration type.
```

Assuming that this entry appears in the task DISKCONTROLLER, another task can call this entry in a variety of ways.

- with a simple entry call:

```
DISK_DRIVER.RESERVE(X);                  -- X is a variable local to the
                                         -- caller of type TRACK_INDEX.
```

This call behaves as a procedure call in the sense that when the call is finally completed, control returns to the statement following the call. But the mechanism is more complex: if DISKSERVER is not ready to accept the call, the caller must wait until DISKDRIVER becomes ready. Only then the RendezVous between caller and callee takes place.

- with a Conditional Entry Call:

```
select
  DISK_SERVER.RESERVE(X);
  -- a possibly empty sequence S of statements
else
  -- a possibly empty sequence T of statements
end select;
```

Now if DISKSERVER is not immediately available to accept the entry call, the call is not made and the sequence T is executed, after which the conditional entry call statement is completed. Otherwise, the entry is called and it behaves as a simple entry call. Then the sequence S is executed. After which the conditional entry call statement is completed.

- with a timed entry call:

```
select
  DISK_SERVER.RESERVE(X);
  -- a possibly empty sequence S of statements
or
  delay D;
  -- a possibly empty sequence T of statements
end select;
```

Here, if the rendez-vous is possible within D seconds, the call is completed as a simple entry call, the sequence S is executed and the timed entry call statement is completed. Otherwise the sequence T is executed and the timed entry call statement is

completed.

Entry families are like:

```
entry TRANSFER (TRACK_INDEX)(THE_OP: in OPERATION;
                             B:      in SECTOR_INDEX;
                             C:      in out SECTOR);
```

where TRACKINDEX is a discrete type, say, INTEGER range 0 .. 199. Then this entry family has 200 members. When a call is made to the family TRANSFER, a specific member must be specified. Apart from this, calls to family entries are like calls to simple entries. For example, if I is a variable of type TRACKINDEX, then a simple call to the Ith entry of TRANSFER is:

```
DISK_SERVER.TRANSFER(I)(X,Y,Z); -- X,Y,Z are actual parameters
                                -- of the appropriate types.
```

Conditional and timed entry calls to family entries are similar to the corresponding type of call to simple entries.

We have seen that a task can make available entries to other tasks. Then these tasks can call those entries in the manner we have described. Now we have to show what the original task does when its entries are called.

The callee task can execute an Accept Statement, something like:

```
accept TRANSFER(NEXT)(THE_OP: in OPERATIONS;
                      B:      in SECTOR_INDEX;
                      C:      in out SECTOR) do
    LOW_LEVEL_DISK_PACK.TRANSFER(THE_OP, NEXT, B, C);
end TRANSFER;
```

This statement accepts a call to the member NEXT of the TRANSFER entry family (NEXT must be an object of type TRACKINDEX). When the callee executes this statement, it checks to see if there is a call to the 'NEXT' member of the family. If there is not, it waits indefinitely for such a call. When the call arrives (or if the caller was already there) a rendez-vous is started during which the caller waits for completion of the call. The TRANSFER operation of LOWLEVELDISKPACK is performed. Then the rendez-vous is completed, i.e. both the entry call statement and the accept statement are completed and caller and callee can continue on their merry ways.

Alternatively the callee can execute a Selective Wait Statement. Here the callee waits not just for a call to one of its entries, but for calls to any of a number of entries. The callee can also choose to accept a call only if it comes within a specified time, or only if no delay is involved.

Here is an example of a Selective Wait Statement:

```
select
```

```

accept RESERVE(A:      in TRACK_INDEX) do
    -- a sequence P of statements
end RESERVE;
or
when RESERVE'COUNT > 0 =>
    accept TRANSFER(NEXT)(THE_OP:  in OPERATIONS;
                             B:      in SECTOR_INDEX;
                             C:      in out SECTOR) do
        -- a sequence Q of statements
    end TRANSFER;
    -- a sequence R of statements
end select;

```

The clause

```
when RESERVE'COUNT > 0 =>
```

is called the Guard of the TRANSFER arm (it will be true if there is no pending call to RESERVE). The RESERVE arm has no explicit guard, and it is treated as if it had a guard that is always true. When the Selective Wait statement is executed, the guards of the various arms are evaluated. If a guard has value true we say that the corresponding arm is open. Then if there are calls to some of the entries of the open arms, one of these calls is accepted (which one we don't know). In the example above a call to RESERVE has precedence over calls to TRANSFER because if a RESERVE call were waiting, the TRANSFER arm would be closed. So if there is a call to RESERVE, it is accepted, the sequence P is executed, after which both the rendez-vous and the Selective Wait statement are completed.

If at the time that the Selective Wait statement is executed there is no pending call on RESERVE, then both the RESERVE and the TRANSFER arm are open. If a call to the NEXT member of the TRANSFER family is pending, it is accepted. The sequence Q is executed, after which the rendez-vous is completed. Then the R sequence is executed and the Selective Wait statement is completed. If instead there is no pending call to the NEXT member of the TRANSFER family, the callee task waits for the first call to RESERVE or to the NEXT member of TRANSFER. This call is then accepted in the manner described above.

Variations of the Selective Wait statement take the form:

```

select
    accept RESERVE .....
    ....
or
    when RESERVE'COUNT = 0 => ....
    ....
else
    -- a sequence U of statements
end select;

```

Here if a rendez-vous cannot be reached immediately, the sequence U is executed and the Selective Wait statement is completed.

```

select
  accept RESERVE ....
  ....
or
  when RESERVE'COUNT = 0 => ....
  ....
or
  delay D;
  -- a sequence V of statements
end select;

```

Now if no rendez-vous is possible within D seconds, the V sequence is executed and the Selective Wait statement is terminated.

Finally, there is the form:

```

select
  accept RESERVE ....
  ....
or
  when RESERVE'COUNT = 0 => ....
  ....
or
  terminate;
end select;

```

The terminate alternative, if taken, terminates the execution of the callee task. The terminate alternative can be taken when no other alternative can be taken. The exact circumstances in which the terminate alternative is taken will be described later.

Here are some additional rules and statements dealing with concurrent tasks.

- Tasks may have different priorities. Assuming that the Ada program is executing on a single processor, then it is not possible for a task of higher priority to be ready but not running while a task of lower priority is running.
- If an exception arises during a rendez-vous and it is not handled therein, it is propagated in both the caller and the callee tasks.
- The Delay statement

```

    delay D;

```

where D is a simple expression of type DURATION [1], suspends execution of the given task for D seconds.

- Task types are possible as are access types that refer to task types. For example:

```

task type BUFFER is
  entry PUT(C: in CHARACTER);

```

```

    entry GET(C: out CHARACTER);
end BUFFER;

.....
B:      BUFFER;           -- a task object of type BUFFER the
type TO_BUFFER is access BUFFER; -- type of pointers to BUFFER tasks
TB:     TO_BUFFER;        -- a pointer to a BUFFER task object.
                               -- initially it is NIL.

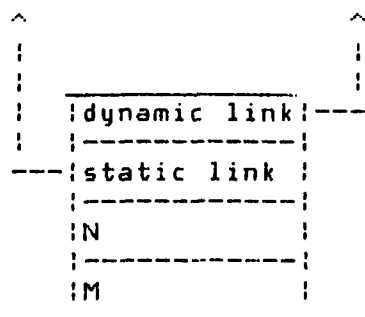
.....
TB := new BUFFER;         -- a BUFFER task is created and asso-
                               -- ciated to TB.

```

[1] DURATION is a predefined fixed-point type. DURATION constants are written as real constants and denote seconds. So 3.142 means 3 seconds and 142 milliseconds.

2.1 THE STORAGE STRUCTURE OF ADA PROGRAMS

In Figure 2.1 appears a sequential program of a block structured language. In this program each time that a procedure is invoked an activation frame (or more simply, just a frame) is allocated on the stack. In the frame there is information for returning to the caller and for re-establishing the conditions that existed therein. The pointer that allows the access to this kind of information is the called the Dynamic Link. In the frame there is information for accessing variables declared in procedures enclosing (textually) the procedure being called. The pointer that allows the access to this kind of information is called the Static Link. In the frame there is also space for the local variables and for the temporary variables of the procedure being called. For example, a simplified form of a frame of the procedure B declared in Figure 2.1 is:



If the procedure B is called and it in turn calls P which calls C, then P, then C, the following calling sequence takes place

B --> P --> C --> P --> C

and the structure of activation frames depicted in Figure 2.2 evolves.

The computation of the Links takes place as follows:

DYNAMIC LINK:

The dynamic link of the callee is set to the address of the frame of the caller.

STATIC LINK:

```

IF the caller and the callee are siblings (i.e. declared in the
   same procedure ) THEN
   the static link of the callee is set to the static link of
   the caller
ELSIF the callee is the son of the caller (i.e. declared
   immediately within the caller ) THEN
   the static link of the callee is set to the address of the
   frame of the caller
ELSIF the callee is the Ith ancestor of the caller (i.e. I=1
   means parent, I=2 means parent of parent ) THEN
   the static link of the callee is set to the address of the
   frame determined by following the static link of the caller
   (I-1) times
END IF;
```

The computation of the static and dynamic links is done each time that a frame is created. The return to the caller is done by following the dynamic link. Access to variables is done in terms of the static links. In an activation of a procedure the addresses of the variables mentioned in the procedure are known at compile time as a pair

<j, d>

where j identifies the frame where the variable is allocated and d gives the position (displacement) of the variable within the frame. For example in the program seen above, the only activation of the procedure B knows its variables N and M as having j equal to 0 (the displacement will depend on the sizes of the objects and it is not relevant to us). These same variables are known in all activations of P as having j equal to 1, and in all activations of C as having j equal to 2. Notice that the first activation of C 'sees' the first activation of P; and that the second activation of C 'sees' the second activation of P but not the first activation of P.

As we are dealing with a block structured language, the various frames can be allocated on a stack, with consequent simplification of the policy needed to reclaim the storage associated to these frames. The fact that the NEBULA Architecture supports Context stacks and the automatic saving of registers and parameters does not substantially affect the arguments presented above. It just reduces the amount of information that needs to be kept on the data stack. For the example seen in Figure 2.2 the situation becomes the one shown in Figure 2.3.

```
procedure B is
  N: ...
  M: ...
  procedure P (X: ...) is
    N: ...
    procedure C is
      Q: ...
      M: ...
      begin
        ...
        P(M);
        ...
      end C;
    begin
      ...
      C;
      ...
    end P;
  begin
    ...
    P(N);
    ...
  end B;
```

FIGURE 2.1

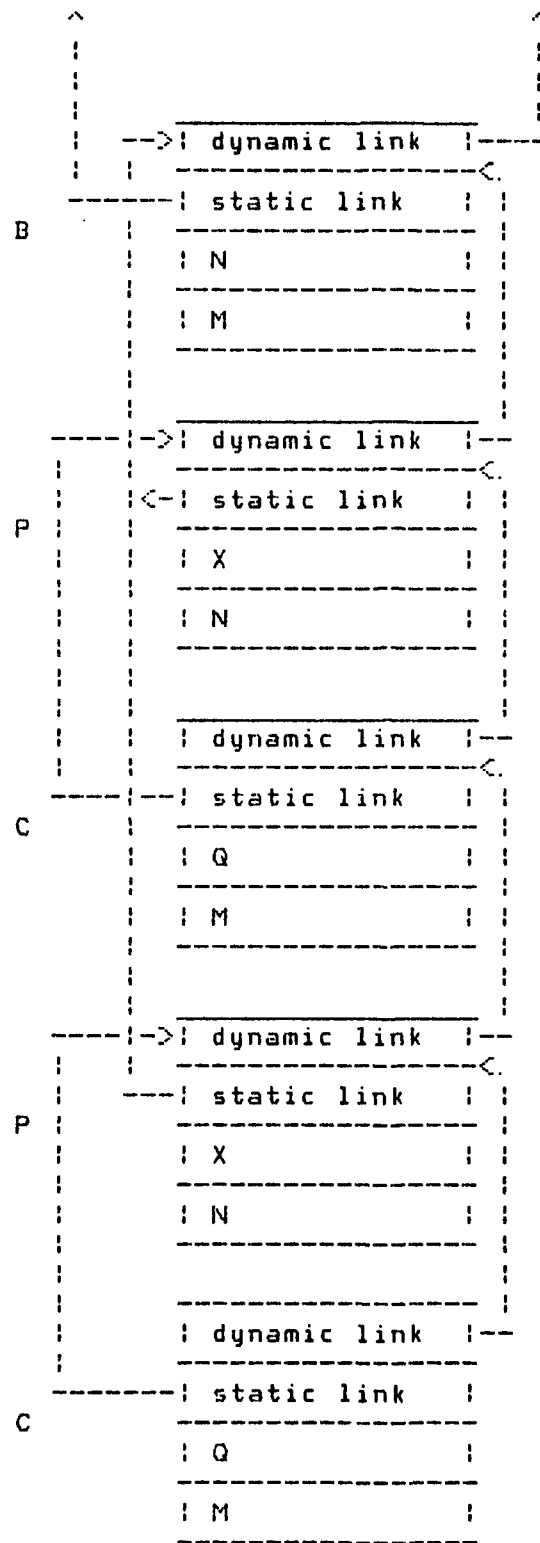


FIGURE 2.2

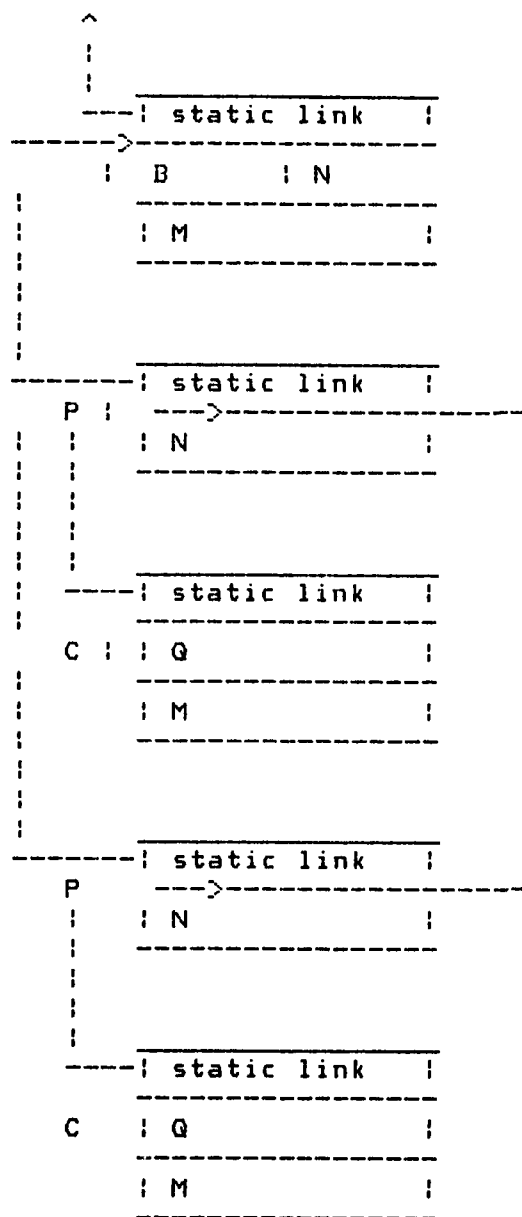


FIGURE 1.3

In Ada storage management is complicated by the presence of dynamic types, that is of types whose objects may remain in existence after the frame where they were allocated has been exited, and by the existence of concurrent tasks.

For dynamic objects we don't actually have to give up the stack discipline. When a dynamic type is declared, a pool of storage is allocated on the stack. When an object accessed by the dynamic type is allocated, space is found for it in the pool of that type. For example, if we have the declarations:

```
type NODE;
type LINK is access NODE;
type NODE is record
    LEFT, RIGHT: LINK;
    VAL:         INTEGER;
end record;
```

at the time that these declarations are processed it is allocated a pool for say n nodes [1]. Later on, when objects of type NODE are allocated with the new operator, they are allocated in that pool, even if the allocation takes place in a frame which is a descendent of the current frame.

The problems posed to the storage management system by the existence of concurrent programs are more serious. These problems are of two kinds:

1. As concurrent tasks execute in interleaved fashion, each allocates and deallocates frames as it wants, without concern for the state of the other. Hence concurrent tasks cannot share the same stack, each must have its own stack. (In the case of NEBULA, of course, two stacks are needed by each task. One is the Context stack and the other is the Data stack). The storage management problem is then how to allocate storage for stacks, given that stacks grow and shrink in a fairly unpredictable fashion. Hence, either each stack must receive a 'maximum' size, or a stack must be able to grow if need be.
2. When a task is declared within another task (the former is said to depend on the latter and the latter is called the master of the former all the objects that are visible in the master at this point are also visible in the dependent task.

Dismissing the first problem by assuming that we can allocate stack segments and possibly expand them without performance penalty, we can provide the needed visibility across tasks as shown in Figure 2.4. There are two fields in each frame where a task is declared, one to head the list of tasks being declared (they are siblings), the other to keep count of the elements in this list. Then at the base of the stack of a task is a pointer back to the frame where it was declared in its master, and a pointer field for linking this task to its

siblings. At compile time each variable is uniquely identified by a triple

$\langle m, j, d \rangle$

where j and d have the same meaning as in block structures languages. m indicates how many master links must be followed starting from the current stack before reaching the correct stack (if m is equal to 0, then the variable is in the current stack). Of course, at run time the access to the variable denoted by $\langle m, j, d \rangle$ will be achieved by carrying out the actions implied by this triple. First, select correct stack using m , then select correct frame using j , and finally choose correct byte using d .

Note that as long as a dependent task is in existence, its master task cannot pop the frame where the dependent task was declared. Otherwise the dependent task would lose its ability to access the variables that are visible to it. This is the reason why in Ada no frame can be exited until all its dependent tasks are terminated. A program unit that is done but cannot exit because of the existence of some dependent task, is said to be Completed.

Of course, alternative strategies for supporting visibility across the Master Task/Dependent Task chasm could be followed. For example, at the base of the stack of the dependent task could have been placed a table with one entry for each task on which this task is dependent (directly or indirectly). This entry identifies the frame of the task where the declaration forcing dependence was made. Then again objects can be represented by triples $\langle m, j, d \rangle$. j and d have the same meaning as before. m instead identifies the entry of the base table that must be used.

At this point we have reached an understanding of how storage can be managed and objects accessed in Ada. We have seen that each task has its own stack (two stacks in NEBULA), that block-structured visibility across tasks could be supported, that the existence of dependent tasks can force the suspension of a task, that dynamic data types need not complicate the storage management of the individual task. We have left unresolved the question of where stacks are actually allocated, and the whole question of what can be done with data objects visible across tasks.

[1] The value of n is chosen by default by the compiler. However the programmer could override this default.

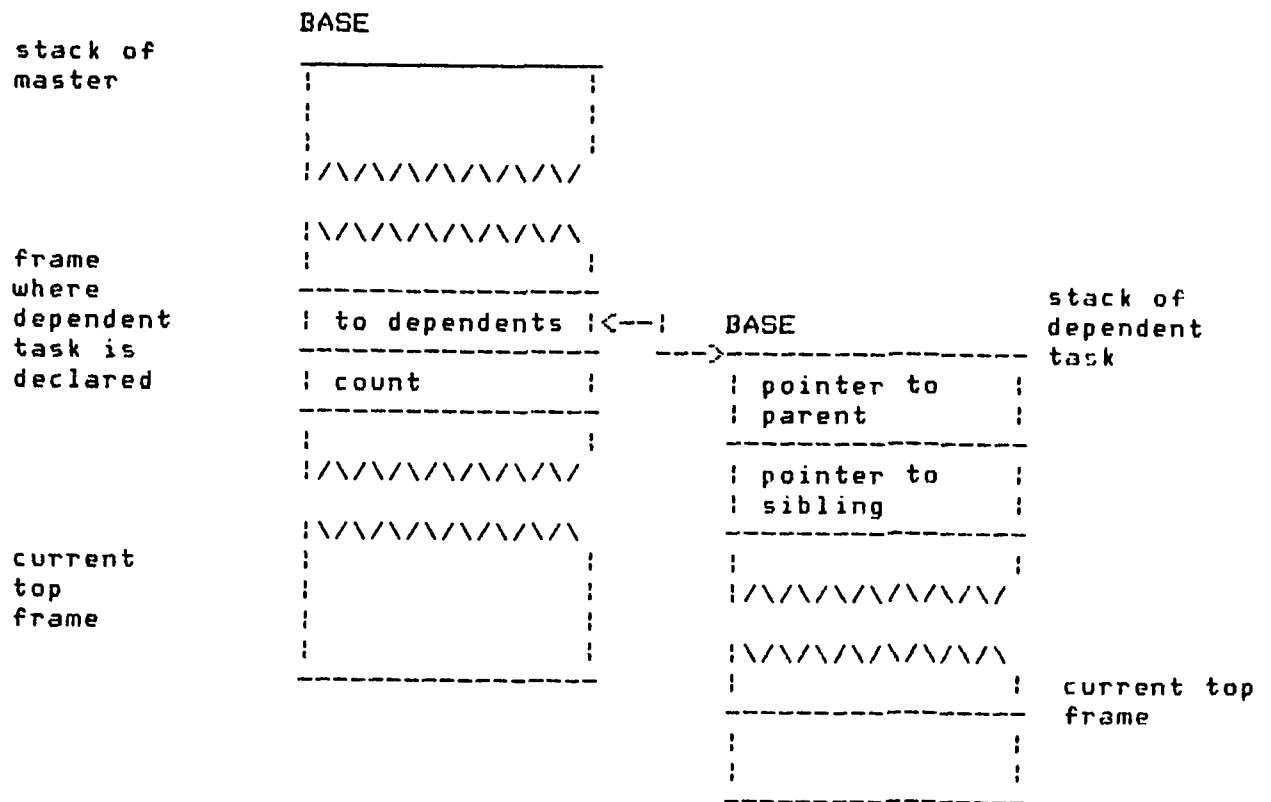
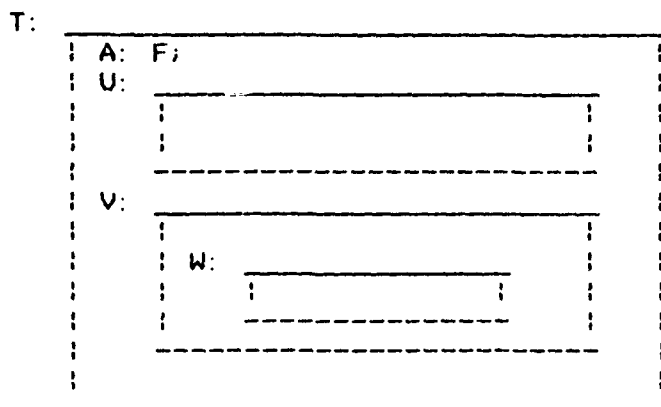


FIGURE 2.4

2.2 SHARING DATA ACROSS TASKS

We have seen that when a task is declared within another task variables declared in the latter are visible to the former. For example, if we have the following situation:



where U and V are tasks declared within task T and W is a task declared within V. Then the variable A of type F declared in T is visible to the tasks T, U, V, W.

We are all familiar with the problems that may arise when data is shared among concurrent tasks. What Ada does to deal with these problems is complex.

Given a task a synchronisation point for this task is when it is activated, or activates another task, or when it executes an Accept statement, or a Selective Wait statement, or some form of Entry Call, or a Delay statement, or an Abort statement, or when the task terminates. That is, a synchronization point of a task is any time when the task invokes some extratask concurrency service[1]. Ada requires that if a task reads (writes) a scalar or access variable that is shared, then in the time interval between the synchronisation points that enclose this access no other task is writing (read or writing) that variable. Programs where this condition is not satisfied are erroneous, i.e. with unpredictable behavior. This regulation makes it possible for tasks to keep local copies of shared scalar and access variables as long as the original is updated (if necessary) at the end of a synchronisation interval where a local copy has been written, and a local copy is updated at the beginning of each synchronisation interval in which it is read before being written.

[1] I exclude Exception Handling from the Synchronisation situations because exception handling is a sequential programming concept.

Ada does not say anything about shared objects that are not of scalar or access type. Even though Ada does not allow local copies of objects that are shared if not of scalar or access types, we will proceed as if what applied to scalar and access objects applied also to all other objects. This means that for any object A of type F:

1. The read operation on A by any task is a logically indivisible operation
2. The write operation on A by any task is a logically indivisible operation

This can be achieved by using a hardware machine architecture where these operations are implemented by uninterruptible instructions, or by assuming that in Synchronisation intervals the reader and writers protocol discussed above will hold.

Going back to our example with the tasks T, U, V, W sharing the object A, we require that any access to A must be completed before any other access to it can be started. In the case that A is of some 'simple' type, the physical operations of access to A are by their nature atomic. For other types, however, an access to A may be done by an uninterruptible operation, in which case serialisation of accesses must be achieved with synchronisation operations.

2.3 MORE ON CONCURRENCY: NORMAL AND ABNORMAL TASK TERMINATION

The Abort statement has form:

abort task_name_1, ..., task_name_n;

For each of the named tasks, if the task is not already terminated, abort will:

1. Mark the task as ABNORMAL;
2. Mark all descendants of the given task, both direct and indirect, as ABNORMAL;
3. For all the tasks mentioned in 1. or 2., if they are suspended in some queue (i.e. they are delayed, or waiting to start a rendez-vous) mark them COMPLETED and remove them from the queue they are waiting in;
4. Mark as TERMINATED all tasks mentioned in 1. or 2. that are not yet activated;
5. For all tasks mentioned in 1. or 2. but not in 3. or 4., they must be marked completed at least by the time that they reach a synchronisation point;
6. For each of the tasks mentioned above which are not terminated, if all their dependent tasks are terminated, mark them as terminated.

Clearly the Abort statement has a radical effect on the tasks it

aborts. If task T has just been aborted, by the time it reaches a synchronisation point it is, if without dependents, terminated. Notice however that if T is in an infinite loop (something like <<INFINITE>>: goto INFINITE;), there is no guarantee that T will be terminated (though it will in any sensible implementation). Any attempt to call an entry of a completed or terminated task raises the TASKINGERROR exception in the caller.

Each task T has two attributes: TERMINATED and CALLABLE.

T'TERMINATED iff T is terminated

T'CALLABLE iff T is not completed, not terminated, and not abnormal.

We can now go back to examine the meaning of the TERMINATE alternative in Selective Wait statements. Assume that a task is waiting in a Selective Wait statement with an open terminate alternative. Then if this task has a master which is complete and whose dependent tasks are all either terminated or waiting in a Selective Wait statement with an open Terminate alternative, then that master task and all its dependent tasks are terminated. Note that in this definition it is not possible for the chosen master to have dependents that are completed but not terminated. Equally impossible is to have, at the time that the master task and its dependents are terminated, any entry call pending on any of their entries.

3.0 AN OVERVIEW OF NEBULA

3.1 THE ADDRESS SPACES OF NEBULA

NEBULA is a virtual memory machine. Its virtual to physical address translation may or not be enabled, under program control. When the translation is disabled, it is still possible to take advantage of the protection features that the Memory Management Unit provides, and to protect differently different areas of physical memory.

3.1.1 THE PHYSICAL ADDRESS SPACE -

The NEBULA Architecture supports a 32-bit physical address. Particular implementations may use smaller physical addresses which will behave as zero extended to 32 bits. (That is, if the actual physical address is 26-bit long, it is treated as if extended with

```

26 27 28 29 30 31
+---+---+---+---+
!0 !0 !0 !0 !0 !
+---+---+---+---+ )

```

The first $2^{*}20$ bytes of physical memory form what is called the I/O SPACE, that is, they are used to address control registers of the I/O controllers and of the processor itself. The top (largest addresses) 2K bytes of the I/O space are reserved for processor control registers. (Readers familiar with the PDP-11 Architecture can compare NEBULA's I/O Space to the PDP-11 I/O Space which is placed in the top 8K bytes of physical memory).

Immediately after the I/O Space, that is, starting at location 100000Hex, is physical memory. The first .5K bytes of this physical memory are taken up by assigned interrupt and trap vectors, or reserved for use by particular implementations of the NEBULA Architecture to carry out Initial Program Loads.

At last, starting at location 100400Hex is the physical memory available to the programmer. The map of the Physical Address Space is recapitulated in Figure 3.1.

3.1.2 THE VIRTUAL ADDRESS SPACE -

A Virtual Address in NEBULA is 32-bit wide. The Virtual Address Space consists of two regions, each $2^{*}31$ byte long, called respectively, User Region and Supervisor Region.

The Supervisor Region is accessible only when the executing code is in Supervisory State.

The User Region is accessible independently of the state of the executing code. The map of the Virtual Address Space is recapitulated

in Figure 3.2. A task, depending on its state, can hence access the whole Address Space, or just the User Region.

NEBULA supports multiple Virtual Address Spaces. Each task has its own Virtual Address Space (of course, the Address Spaces of distinct tasks could be identical). The Virtual Address Space of distinct tasks share the same Supervisory Region, but have distinct User Regions in the sense that each task can have its own mapping from the User Region of the Virtual Memory to the Physical Memory.

3.1.3 VIRTUAL TO PHYSICAL ADDRESS TRANSLATION -

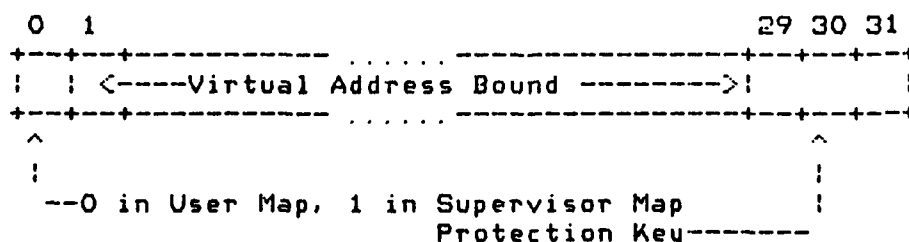
The translation of virtual addresses in the User Region and in the Supervisor Region are done independently of each other. In either case the same method is used. Hence it will be sufficient for us to consider the case of the User Region.

The User Region is divided into Segments. The number of segments in a User Region is not fixed, so that different User Regions can have different numbers of segments. In each implementation of NEBULA at least 16 segments will be supported.

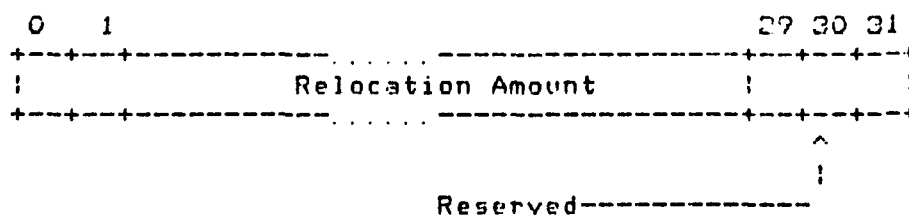
The size of a segment is not predetermined, in the sense that segments of different sizes can be defined and a segment could be made as large as allowed by the underlying Physical Memory.

The translation from the User Region to the Physical Memory is controlled by a Segment Map accessed from a special register, called the User Memory Map Pointer (UMMPP) Register. The Segment Map is preceded by a 32-bit word specifying the number of entries in this map.

Each entry in the Segment Map consists of two 32-bit words. The first word has form:



and the second has form:



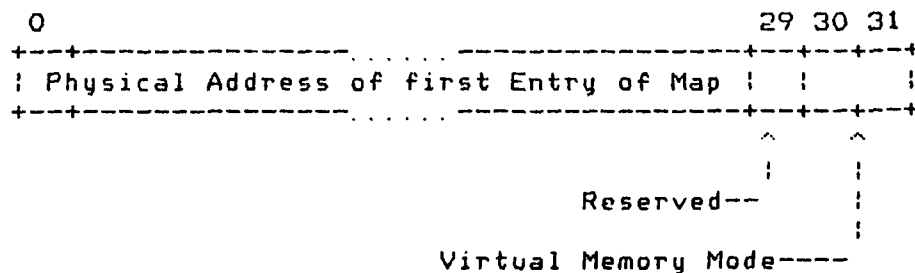
Successive entries in the Map must have increasing virtual address bounds.

Given a Virtual Address in the User Region, to it is associated at most one entry in the User Region Map. This entry, if it exists, it is the first one whose Virtual Address Bound is not less than bits 1..28 of the given virtual address. All entries of a Map must be aligned on double word boundaries.

The values of the Protection key have the following interpretation:

- 000 No access is allowed. This mode will be usually used for swapped out segments
- 001 Data Read Only
- 100 Execute only. This mode allows the reading of literals
- 101 Execute or Data Read
- 010 Data Read or Data Write
- 110 Reserved
- 011 Context only. It is the mode of Context Stacks (see 3.3)
- 111 Reserved.

The User Memory Map Pointer has form:



The Virtual Memory Mode has the following interpretation. If it has value:

- 00 It is as if the machine had no virtual memory, or if you prefer, the map is the identity map,
- 11 The entry corresponding to a given virtual address is determined. The the first 29 bits of the physical address are obtained by adding (overflow, if any, is discarded) the first 29 bits of both the virtual address and the relocation amount of the entry. The last 3 bits of the physical address are the same as the last 3 bits of the virtual address. The legality of the access is checked using the Protection Key.
- 01 The entry corresponding to a given virtual address is determined. The corresponding physical address is the same as the given virtual address. The legality of the access is checked using the protection key of the entry.
- 10 Reserved

3.1.4 MEMORY MANAGEMENT TRAPS -

A number of traps are associated to errors that can occur during accesses to memory.

If the physical address used is not implemented, a trap (vector at 100014Hex) is invoked with as only parameter the guilty address.

The remaining traps are Memory Management Traps, i.e. they are associated with errors that can arise in the virtual to physical address translation. All Memory Management Traps share the same vector (at 1000000Hex) and have 4 parameters:

- ?1 The address that gave origin to the fault
- ?2 The address of the operator of the instruction that was executing at the time of the fault
- ?3 The segment number, if any, associated to the address (?1 above) causing the fault
- ?4 The fault code. It is a byte with values:
 - 1 Illegal reference to Supervisor Region
 - 2 No segment was found for this address
 - 3 Protection violation
 - 4 Privilege Violation

As the reader can see, the information provided by these faults is bountiful and should allow for graceful recovery from Memory Management Traps.

3.1.5 HARDWARE SUPPORT FOR REPLACEMENT POLICY -

The NEBULA Architecture has no hardware to support the implementation of any policy for choosing the segments to be removed from main memory when main memory is full and a segment fault has taken place. In particular, contrary to most other machines, no Use Bit (set each time a segment is accessed), or Dirty Bit (set each time a value contained in a segment is modified) is associated by the Architecture to each segment. Of course, particular implementations of the Architecture are free (and likely to) add such bits.

3.1.6 SETTING AND MODIFYING VIRTUAL ADDRESS SPACES -

We have seen that the Address Space currently in use is characterised by the data structures shown in Figure 3.3.

The User Map and Supervisor Map are created as any other data structure. When they are fully set up, they are activated by loading the corresponding Map Pointer Register with these addresses.

The Supervisor Memory Map Pointer is likely to remain unchanged throughout an up-time session of the machine. This is so since the Supervisor Region is shared by all Address Spaces. The initial

loading of the Supervisor Memory Map Register is done by using its address in the I/O region.

The User Memory Map Pointer is loaded and saved (together with the Task Context Pointer that we shall encounter later) using the Load Task and the Store Task instructions. As tasks take turns in sharing the processor, changes in the content of the User Memory Map Pointer are going to be frequent.

3.1.7 SOME SPECIAL MEMORY MANAGEMENT INSTRUCTIONS -

NEBULA has some instructions that are very convenient for manipulating Virtual Address Spaces.

The Repent instruction is available for modifying an entry of the User Map of the currently executing task, or an entry of the Supervisor Map. This instruction can be used, for example, in the following two cases:

- In the case of a segment fault, after the needed segment has been placed in main memory, we can use the Repent instruction to patch the corresponding Map entry to reflect the new situation.

- The Repent instruction can be used to map a segment of the User Region of the current task into different areas of physical memory at different times.

The Map instruction, given the physical address of a Map and a Virtual Address, it interprets the Virtual Address using the Map and returns the corresponding segment number and physical address.

3.2 PROCEDURES IN NEBULA

The notion of 'Procedure Call' is ubiquitous in NEBULA. Many control structures that in other architectures have separate mechanisms are reinterpreted in NEBULA as procedure calls. In a very real sense one can say that at all times in NEBULA the executing code has been invoked with a procedure call.

A procedure in NEBULA is not just a sequence of instructions, supported by simple instructions for invoking it and returning from it:

- Each activation of a procedure has available up to 16 registers (the program counter is register 0) that are treated, with the exception of register 1, as local variables of the activation. That is, when we call a procedure the registers of the caller are saved in the 'context stack' together with the Processor Status Word (PSW), and the called procedure is given essentially a new set of registers. The initial content of these registers (with

the exception of registers 0 and 1) is undefined. Upon return to the caller, the content of the callee's registers is lost and the registers and PSW of the caller are re-established. The nature of Context Stacks will be dealt with more in detail in section 3.3. For now it suffices to know that the machine maintains at one time two context stacks, the Kernel Context Stack and the User Context Stack. The former is used when the machine is in Kernel mode, the latter when the machine is in User mode. The same Kernel stack is used at all times. Instead at different times different User Context Stacks can be used as they are in one-to-one correspondence with tasks.

Parameter passing is implicit to the call mechanism. That is, if we want to call the procedure P with arguments A1, A2, ..., An ($n \leq 256$), the call is effected with the single machine instruction:

CALL P, A1, A2, ..., An

Within this activation of P these parameters are accessed with the names:

?1 ?2 ... ?n

These names are local to this activation of P and consequently are not visible in any procedure called by P. If we want, say ?3 to be visible in a procedure Q called by P, either we must use ?3 as a parameter in the call to Q, or we must copy ?3 to some area accessible from Q.

The code of each procedure includes as a prefix a descriptor. This descriptor will indicate the maximum number of registers needed by this procedure, whether the procedure expects a variable or fixed number of parameters, and, if fixed, how many. It also indicates if exceptions raised during an activation of the procedure should be handled by a system wide exception handler or instead by a handler provided by the user.

Of course 'procedure calls' are used to call procedures. But many more control mechanisms are interpreted as procedure calls.

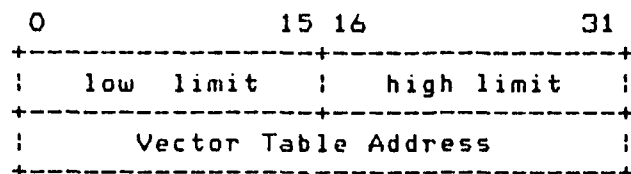
Interrupts are interpreted as procedure calls invoked by the hardware to the appropriate interrupt handler. All interrupt handlers are treated as kernel mode procedures, hence they use the Kernel Context Stack.

Traps are also interpreted as procedure calls made by the hardware. They again use the Kernel Context Stack. The Trap procedure call will have a number of parameters. Their exact number and nature is dependent on the trap being considered. A number of traps are defined in the NEBULA Architecture. To each trap is associated a trap vector, i.e. a location pointing to the handler to be invoked.

Supervisory Calls, again, are procedure calls. A Supervisory Call instruction has form:

SVC index, p1, ..., pn

The NEBULA Architecture has two special registers with addresses in the I/O space that are used in the interpretation of this instruction. These two registers have the form:



where the second register has a value pointing to a structure, the SVC table, of the form shown in Figure 3.4. When the SVC instruction is executed, a vector is chosen in the SVC Table. This vector is treated as the physical address of the procedure to be called. The arguments $p_1, \dots, p_n$ are given as parameters to this procedure. The SVC call does not change the current mode and priority. The vector determined by the call specifies if the called procedure should or not be executed in Supervisory State and if with or without privilege.

Finally, also the dispatching of a task takes the form of a procedure call. The caller task becomes suspended. The dispatched task continues from the conditions it had when last suspended.

Even though, as indicated above, interrupt handlers, trap handlers, and tasks are invoked as procedures, usually they do not terminate as procedures by performing a return instruction. Calls and returns are not necessarily matched. For example, task A may 'call' task B, which may 'call' A, etc. without any intervening return operation.

As a final observation we note that NEBULA supports the notion of procedure call, not that of function call. Function call instructions would be desirable for the implementation of languages like C and LISP where procedures are treated as special cases of functions.

3.3 CONTEXT STACKS

In the PDP-11 a register, R6, acts as stack pointer. When a procedure is called the return address (and the PSW) is pushed into the stack. Upon return this address (and the PSW) is popped from the stack and moved to the program counter to continue the execution of the calling program. This same stack is used to save registers, to pass parameters, and to allocate the temporary variables of the calling and called procedures.

Some of the activities that on the PDP-11 are explicitly carried out by the program, pushing and popping the stack, are instead implicitly carried out by the hardware using a Context Stack. In this Context Stack are implicitly saved return addresses, PSWs, Registers, and are are implicitly allocated the parameters of calls. The allocation and deallocation of temporary variables is still under program control and is to be done in a separate Data Stack. For this Data Stack we need a stack pointer. This can be done by using Register 1 which, contrary to other registers is not saved and made undefined when a procedure is called.

The NEBULA Architecture provides two registers, The Kernel Context Pointer (KCP) and the Task Context Pointer (TCP). The procedure that is currently executing will use the Context Stack pointed to by one of these pointers. It will use TCP if the current procedure has Task mode. (The mode of a procedure is specified by its descriptor).

Each Context Stack should be implemented as a segment with appropriate protection key. The Context Pointer for this Context Stack should be initially set to the address of the first byte past (higher than) the segment.

The KCP Register should be set only once at start-up time. Then the same Kernel Context Stack will remain in use across all procedure calls. Instead different tasks have different Task Context Stacks. When a task is initiated it is given a Task Context Stack and the TCP is appropriately loaded. Then each time a new task is dispatched, the Context Pointer of the dispatching task is saved and the TCP register is reset to the value saved for the task being dispatched.

3.4 THE PROCESSOR STATUS WORD

The Processor Status Word (PSW) maintains important information about the current status of the processor and of the executing code. The principal information, for the purposes of our discussion, is maintained in the fields:

Kernel/Task Mode: This bit field, if set, states that the context Stack currently in use is the Kernel Context Stack; otherwise it is the Task Context Stack.

Supervisor/User State: This bit field, if set, states that the current virtual address space consists of both the User and Supervisor Region; otherwise it consists of just the User Region.

Privilege Condition: This bit field, if set, states that privileged instructions can be executed and privileged memory segments accessed; otherwise they cannot.

These bit fields are independently controllable to give all 8 possible value patterns. They form what we call the Composite State of the program. We will, in the next session, examine how these bits can be modified.

Other significant fields in the PSW are:

Last Mode: Bit field that is set to the value of the previous Kernel/Task mode.

Priority: A 5-bit field that specifies the current

priority of the processor.

Base of Context Stack: Bit field that is set if the current Context Stack contains only the current procedure activation.

3.5 CONTROLLING THE COMPOSITE STATE OF A PROGRAM

We have called Composite State of a program the combination of the values of the Kernel/Task Mode, Supervisor/User State, and Privilege Condition of the PSW. The Composite State controls the essential features of the machine that are available to a program: the Context Stack it uses, the Address space accessible to it, and the legality of using privileged instructions and segments. In the following we examine how transitions among Composite States can take place.

As there is a Load PSW instruction in NEBULA, it is possible to go from any privileged composite state to any composite state. However, the more useful transitions do not require the use of the Load PSW instruction.

As we have seen, all sorts of control mechanisms are reduced in NEBULA to procedure calls. However, the way the calls are actually made affects the way that transitions among Composite States take place.

- o The Call instruction does not affect the Composite State.
- o The CALLU instruction changes only the privilege condition of the composite state. No matter the current value, it sets the Privilege Condition to no privilege.
- o Supervisory Calls, Interrupts and Traps change the composite state in a way that is essentially independent of the Composite State at the beginning of the transition. The Supervisor/User state is set to bit 0 of the address of the invoked procedure (i.e., procedures in the User Region can only access the User Region; and procedures in the Supervisor Region can access both the User and Supervisor region).
The Privilege Condition is set to bit 31 of the address of the invoked procedure (in the case of SVCs, this value is ORed with the current privilege condition).
The Kernel/Task Mode is set to Kernel in the case of traps and interrupts; it is left unchanged in the case of SVCs.
Interrupt calls set the priority to the priority of the interrupting device. SVCs do not change the priority.
- o Task initializations act very much like the Load PSW instruction in that the PSW of the task being initiated is fully specified as part of the initialization.

- o Task dispatching behaves as a Return instruction, not as Procedure Call: the conditions existing when the task was suspended are re-established.

Execution of the Return from Procedure instruction has different behavior depending on the state of the current activation on the Context Stack.

If this activation has its Base marked (it is the initial activation on this Context Stack), on the basis of the PSW field Previous Kernel/Task Mode, after elimination of the current context, a new Context Stack is chosen (it might be the same if the Previous and Current Mode are the same) and its top activation is re-instated. Consequently its Composite State is adopted.

If this Context does not have its Base marked, it is removed and the previous one is re-instated. The Kernel/Task Mode is left unchanged. The Supervisor/User State and the Privilege Condition are reset to the values they had in the new context.

3.6 TASK ORIENTED INSTRUCTIONS

NEBULA has instructions specially dedicated to task manipulation. But first, what is a task? According to Dennis and Van Horn [DVH] a task is:

"... a locus of control within an instruction sequence. ...

That abstract entity which moves through the instructions of a procedure as the procedure is executed by a processor."

A task has an identity and state supported by the operating system and/or the hardware. This support allows the interleaving of the execution of multiple tasks on a single hardware supported execution sequence.

When we program tasks on NEBULA, to each task we associate a two word control block of the form

```

+-----+
| Value of Task Context |
| Pointer Register      |
+-----+
| Value for User Map    |
| Register              |
+-----+
```

The address of this block can serve as identifier of the task associated to that block. Each time the task is activated the first word of the control block is loaded into the Task Context Pointer Register, and the second word of its control block is loaded into the User Memory Map Register. Then this task is dispatched using the top context of its Task Context Stack. As part of this single instruction the context of the task executing this instruction is saved in its Task Context Stack. A separate instruction is used for saving the current value of the control block of the dispatching task.

In other words, for a task to give up the processor and dispatch another task it needs to execute 3 instructions:

1. The Store Task instruction, which, given the address of a Control Block, stores there the current values of the Task Context Pointer Register and of the User Region Memory Map Register.
2. The Load Task instruction, which, given the address of a control block, obtains from it new values for the Task Context Pointer Register and for the User Memory Map Register.
3. The Start Task instruction which starts executing the top context of the Context Stack specified as a parameter of the instruction (the parameter indicates if to use the Task Context Stack or the Kernel Context Stack).

A variation of this instruction will precede the above action by discarding on the specified context stack of all contexts up to and including the first one with its Base bit set.

In either case a specified exception could be raised in the task being dispatched.

Another kind of instruction is available for performing the initial activation of a task. This is the Initiate Task instruction. This instruction has two parameters. The first is the address of the procedure which is the code of the task being initiated. The second provides bits 0:15 of the PSW for that task.

A context with its Base bit set is allocated for the task being activated on the Context Stack specified by bit 0 of the PSW. As part of this instruction the context of the task performing the instruction is saved.

3.7 THE NEBULA MODEL OF TASKS AND OF THE TASKING KERNEL

The NEBULA Architecture supports the basic notion of a kernel, i.e. of code that supports user tasks and their interactions. At a first glance the software architecture supported has the form depicted in Figure 3.5.

In this architecture each task has:

1. An address space of the form

+-----+	Controlled by User Memory Map.
User	It is different in different
Region	tasks.
+-----+	
Supervisor	Controlled by Supervisor Memory
Region	Map. It is the same for all
+-----+	tasks.

2. A Task Context Stack, where the contexts of this task are saved.

3. A Data Segment, used for the dynamic data structures used by this task.
4. A Code Segment, used for the code executed by this task.

All of these areas can be determined by examining the hardware control block of this task.

The Virtual Address Space of distinct tasks share the Supervisor Region and have distinct User Regions. However it is easy to have two User Regions sharing the same physical segment(s). It suffices to set segments in the memory maps of the two User Regions to point to the same physical address(es).

No instruction supports directly semaphore operations, or spin-locks, or other entities used in the synchronisation of tasks. However a Compare and Swap instruction is available for the implementation of synchronisation mechanisms.

The thread of control of an executing task can be changed because of synchronous traps, or of asynchronous interrupts, or of Supervisor calls.

Supervisor calls do not change the current mode, hence they do not change the current Context Stack. They may change the address space to include the Supervisor Region, and they may change the Privilege Condition to allow the execution of privileged instructions. Supervisor calls do not change the identity of the current task or its priority, they change what the current task can do, to include actions that normally are thought of as belonging to the kernel. Though the identity of a task is not changed by the execution of a Supervisor Call, some may prefer to think of the changes that take place as the transmutation of the given task into a more powerful alter-ego, into an 'uplifted version of the given task. From the point of view of the task that executes a Supervisor Call, this call is undistinguishable from a regular procedure call in the sense that normally after some action is done, control continues with the instruction following the Supervisor Call.

Traps change the mode of the executing program. The Kernel/User mode is set to Kernel and on the Kernel stack is created a context with its Base bit set. Privilege is not changed. Priority is set to its maximum value (1F Hexadecimal).

The State will be User or Supervisor depending on the most significant bit of the procedure invoked by the trap. Again the identity of the running task is not changed by a trap.

Interrupts behave just like traps with two differences, the obvious one of being asynchronous instead of synchronous, and the fact that the priority of the procedure called by the interrupt will have its priority set to the priority of the interrupting device.

Traps and Interrupts use the Kernel Context Stack and their actions are nested.

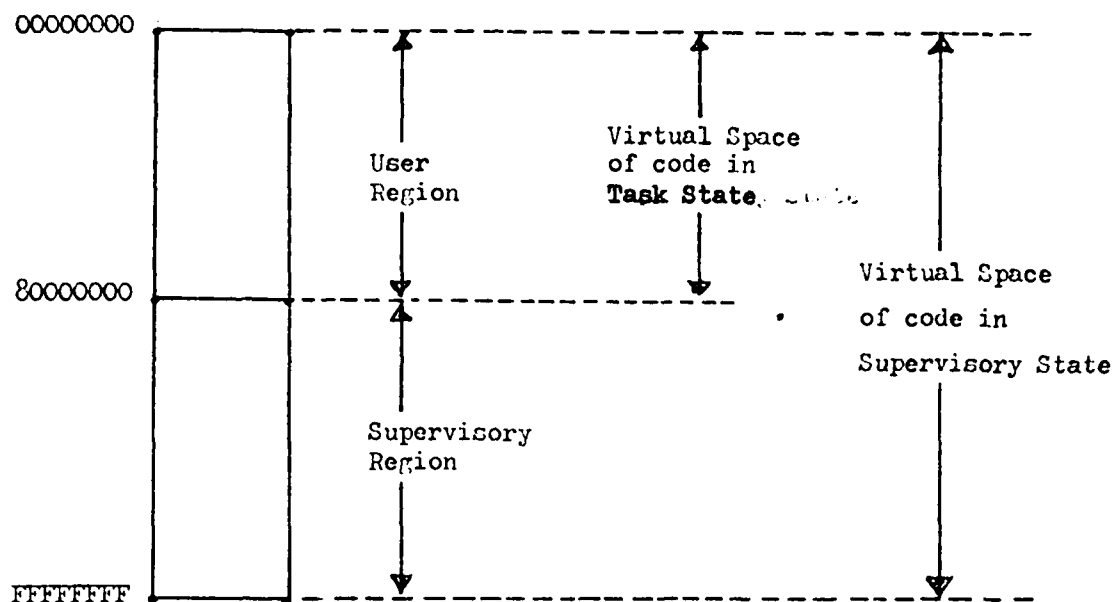


Figure 3.1

A Virtual Address Space

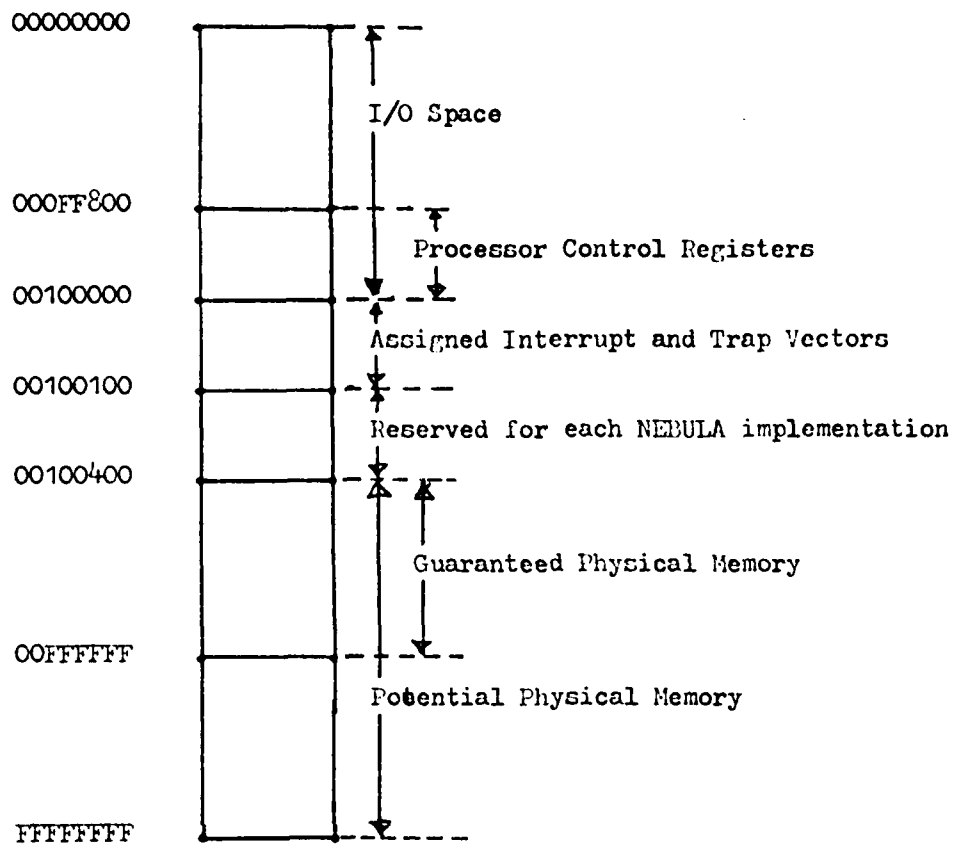


Figure 3.2
The Physical Address Space

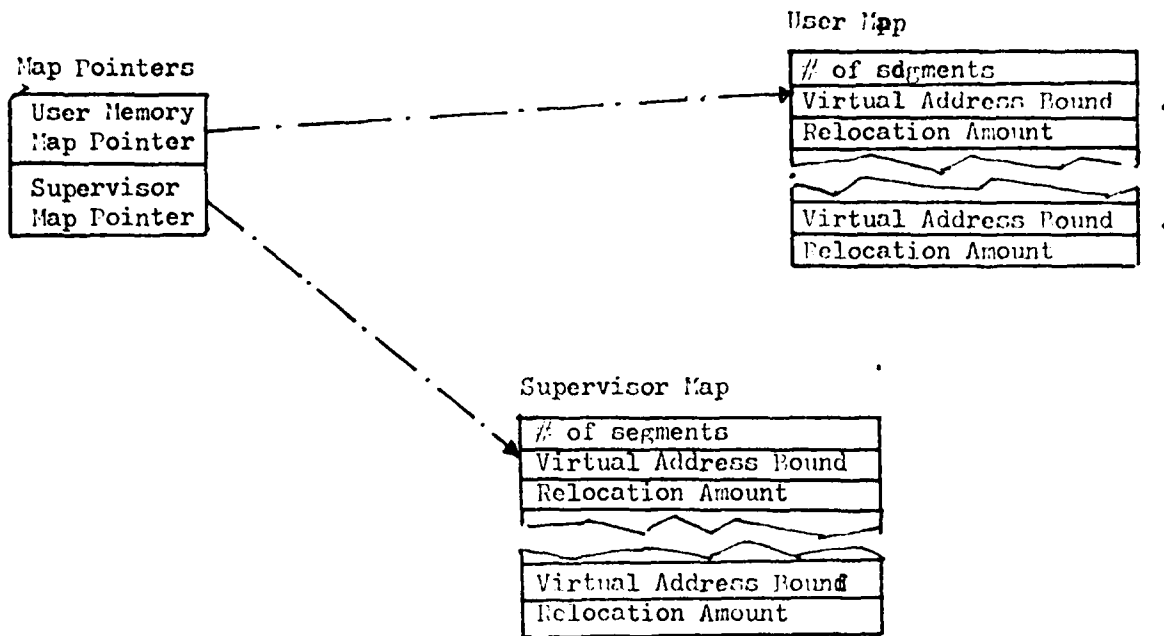


Figure 2.3
Memory Management Maps and Registers

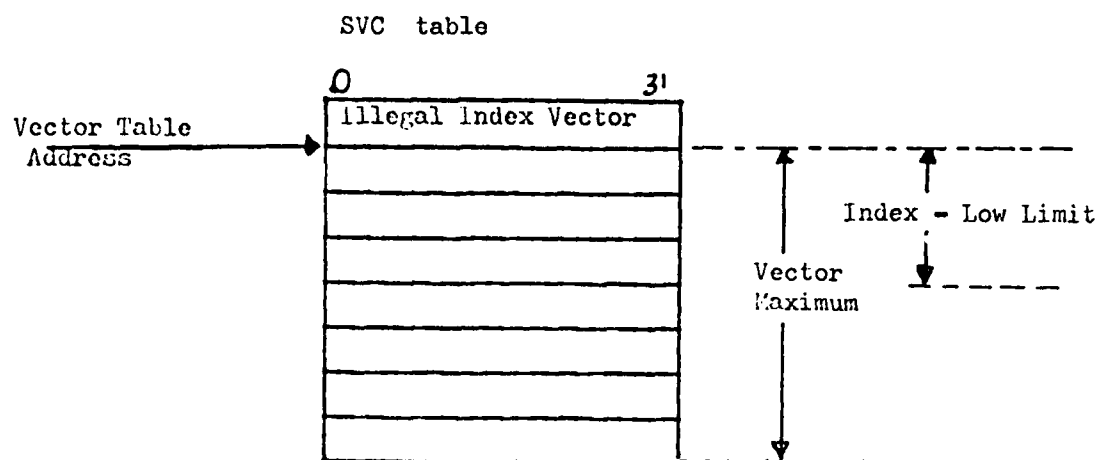


Figure 3.4

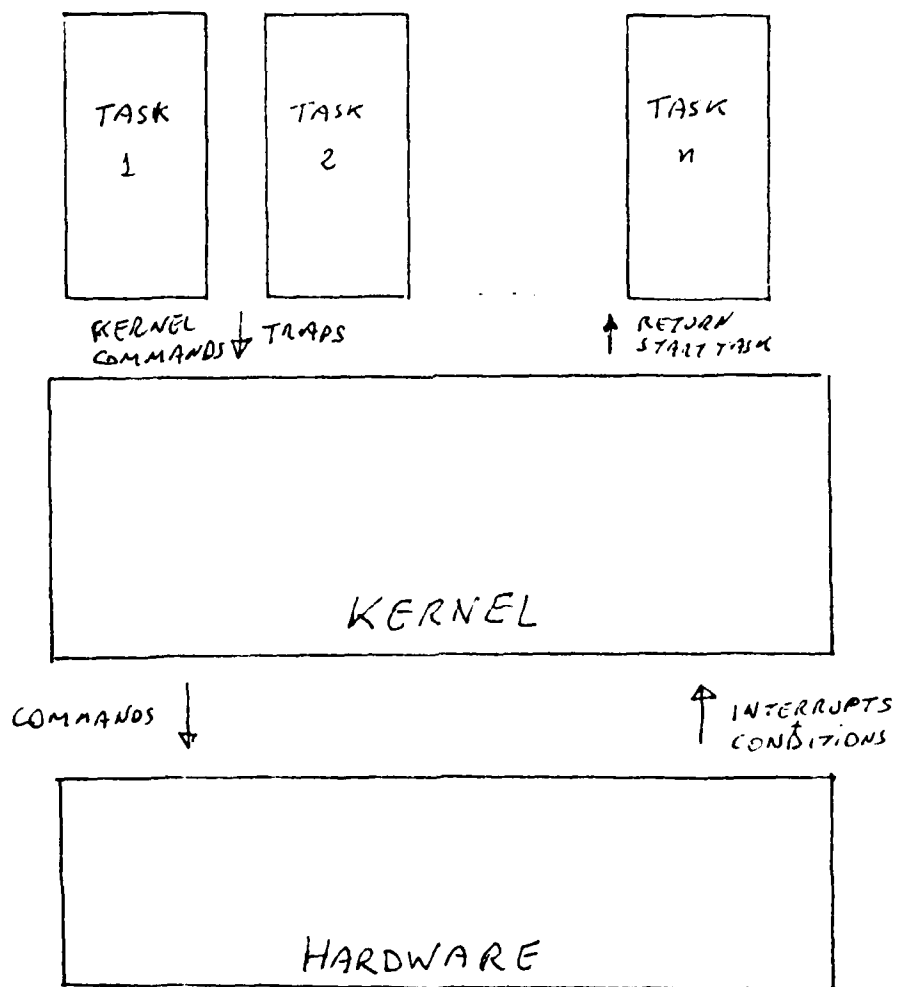


FIGURE 3.5

4.0 A CONCURRENCY KERNEL FOR ADA

The first impression upon reading about Ada tasks and their interactions it is to assume that very complex data structures and control structures are required for their implementation. After a while one comes to the realisation that things are not as difficult as they seem at first.

We now introduce some very limited extensions to what is required for sequential programming in Ada. We show how these extensions can be used to implement the tasking mechanisms of Ada. Later we will consider the implementation of these extensions on the NEDULA Architecture.

All that follows is written under the assumption that all the tasks of a program share the same address space. Whether this address space is virtual or not is irrelevant to our discussion. This assumption of a single address space [1] simplifies the problem of addressing shared variables and of accessing the actual parameters of entry calls. Under this assumption addresses mean the same in all tasks. This assumption limits the protection possible among concurrent tasks and makes relocation of segments in this single address space more difficult.

4.1 TASK CONTROL BLOCKS

For each task we allocate 3 areas:

- . The Data Area: a static area where are allocated all data structures that exist throughout the life of the task;
- . The Context Area: an area used for the context stack of this task and
- . The Stack Area: an area used by the normal stack of this task

We do not assume the existence of a separate Code Area for each task. Instead we assume that a whole Ada program has a single Code Area [2].

When a task T creates a dependent [3] task V, it creates for V a permanent Hardware Context Block (HTCB) [4] which remains in existence as long as V can be named in some intertask action [5]. In the NEDULA Architecture this control block contains the Memory Map Pointer and the Context Pointer.

-
- [1] Different Tasks may still have different memory maps with different protection rights. Further, the supervisor/user distinction differentiates between the areas accessible at different times.
 - [2] We leave unspecified the relationship between this notion of "Area" and NEDULA's notion of a "Segment".

Additional information about a task appears in four other Task Control Blocks:

- . the Permanent Task Control Block (PTCB), which remains in existence as long as the task is not terminated,
- . the Static Task Control Block (STCB), whose existence is required only while the task is activated and not terminated,
- . the Dynamic Task Control Block (DTCB), of which there may be a copy in each activation frame of the task, and
- . The Rendez-vous Task Control Block (RTCB), of which there is a copy for each Rendez-Vous in which the task participates as a callee.

An example of information maintained in the PTCB is the status of the task. Examples of information maintained in the STCB are the indication of unavailability for the task to accept calls on a particular entry and the count of calls pending on an entry. Examples of information maintained in the DTCB are means to account for and access the tasks that are dependent on this activation frame. Examples of information kept in each RTCB are the priority that the task had before the rendez-vous, and indications of the entries calls that can be accepted.

Though the information maintained in the five task control blocks are different in role and in life expectancy, we will do as if all the information they contain were actually maintained in a single Task Control Block (TCB). In Figure 4.1 appears an Ada package that describes the information maintained in a TCB. This package makes appeal to the notion of "queue" which is examined in more detail in section 4.2.

-
- [3] If within block B we declare task V, then V depends on B. If in B we declare the access type P to tasks of task type W, then whenever a task is created through a sequence of actions like:

X: P;

X.all := new W;

the task so allocated is dependent on B.

Then a task V is dependent on a task T if it is dependent on one of the blocks of the task T.

- [4] We call this Control Block "Hardware" because it is directly supported by the NEBULA hardware in the LOAD TASK and STORE TASK instructions.
- [5] This may be well after V has terminated execution, since V can be named as long as the block on which it is dependent is not exited.
-

```

with QUEUE; use QUEUE;
package TASK_CONTROL_BLOCKS is
  type AREA_DESCRIPTOR is
    -- undefined; it is the type of the
    -- descriptor of the Context Area, Data
    -- Area and Stack Area.
  type AT-AREA-DESCRIPTOR is access AREA_DESCRIPTOR;
  type TASK_STATUS is (CALLABLE, COMPLETED, ABORTED, TERMINATED);
  type RESULT_CODE is new NATURAL;
    -- It describes what happens during a rendez-
    -- vous. It can take the following values:
    -- 0    an exception was raised during
    --      rendez-vous
    -- 1    the call was successfully completed
    -- 2    a delay alternative was taken
    -- 2+i  a call to the ith entry of this
    --      task was accepted and successfully
    --      completed.
  WITH_EXCEPTION: constant RESULT_CODE := 0;
  SUCCESS:         constant RESULT_CODE := 1;
  WITH_DELAY:      constant RESULT_CODE := 2
  CALLED_i:        constant RESULT_CODE := 2+i;
  type EXCEPTION_CODE is
    -- SYSTEM defined integer type.
    -- This code represents the exception that
    -- took place during a rendez-vous. In the
    -- NEBULA architecture this quantity is
    -- well defined and retrievable after an
    -- exception with the ECODE instruction.

  type TCB;
  type AT_TCB is access TCB;
  ---
  type TCB is record
    MEMORY_MAP_POINTER: SYSTEM.ADDRESS;
    CONTEXT_POINTER:    SYSTEM.ADDRESS;
    PERMANENT_PRIORITY, -- priority permanently associated to task
    PREVIOUS_PRIORITY,  -- priority in previous activation frame
    CURRENT_PRIORITY:   -- priority in current activation frame
    SYSTEM.PRIORITY;

    ---
    CONTEXT_AREA_POINTER,
    DATA_AREA_POINTER,
    STACK_AREA_POINTER:  AT-AREA-DESCRIPTOR;
    MAIN_ENTRY:          SYSTEM.ADDRESS;
    -- address of where in code area this task
    -- starts execution.

  ---
  STATUS: TASK_STATUS := CALLABLE;
  ---
  WAITING_LIST: LINK;
    -- When this task is waiting in a queue it
    -- is linked into this queue by using this
    -- field

```

FIGURE 4.1 (Cont.)

```

DELAY_LIST:                                LINK;
-- When this task is waiting because of a
-- DELAY statement, it waits in the timer
-- queue using this link

---
SIBLING_LIST:                              LINK;
-- All the siblings of this task are linked
-- together using this link.

CHILDREN:                                  ANCHOR;
-- The anchor of the list of all the
-- dependent tasks.

WHERE_CALLING:                             AT_ANCHOR;
-- The address of the anchor of the list
-- (queue) where this task waits if unable
-- to complete a call.

---
NUMBER_OF_CHILDREN:                        NATURAL := 0;
-- The number of Dependent tasks of this task

NUMBER_OF_CHILDREN_AT_TERMINATE: NATURAL := 0;
-- The number of dependent tasks of this task
-- that are waiting at a Selective Wait
-- statement with a Terminate alternative.

---
RESULT_IS:                                RESULT_CODE;
-- It reports what happened in the last
-- rendez-vous of this task. It is used by
-- the task when acting as callee.

RESULT_WAS:                               RESULT_CODE;
-- It has exactly the same meaning as
-- RESULT_IS but now it is intended for
-- the caller.

EXCEPTION_IS:                             EXCEPTION_CODE;
-- It indicates to the caller of an entry,
-- if there was an exception, what
-- exception it was.

TASK_BEING_CALLED:                         AT_TCB;
ENTRY_BEING_CALLED:                       NATURAL;
-- It indicates the entry being called in a
-- Selective Wait.

---
WITH_TERMINATE_ALTERNATIVE:               BOOLEAN := FALSE;
-- True iff this task is waiting at a
-- Selective Wait statement with a
-- Terminate alternative.

WITH_DELAY_ALTERNATIVE:                   BOOLEAN := FALSE;
-- True iff this task is waiting at a
-- Selective Wait alternative, or at a
-- Timed Entry Call

DELAY_COMPLETED:                           BOOLEAN := FALSE;
-- True iff a Delay was requested, and, if
-- it expires, it can be accepted.

```

FIGURE 4.1 (cont.)

```

MUTEX:                                SEMAPHORE;
-- It serializes all interactions with this
-- task.

GATE:                                array (1 .. NO_OF_ENTRIES) of BOOLEAN
:=                                  (others => FALSE);

WLIST:                                array (1 .. NO_OF_ENTRIES) of ANCHOR;
-- GATE and WLIST control access to the
-- entries of this task. They have one
-- component per entry. The GATE(i) will
-- be true iff a call to the ith entry can
-- be accepted. WLIST(i) is the anchor
-- of the list where callers to ith entry
-- can wait.

end record;
end TASK_CONTROL_BLOCKS;

```

FIGURE 4.1

4.2 SINGLY LINKED QUEUES AND SEMAPHORES

In this Section we implement queues and semaphores using:

- . The sequential facilities of Ada,
- . an uninterruptible procedure COMPARE-AND-SWAP that we know to be supported at the machine level in NEBULA by a single instruction, and
- . three procedures, GIVE-UP, GIVE-UP-WITH-SUBSTITUTE and READY-ENQUEUE, defined in the package KERNEL-INTERFACE and discussed in Section 4.4.

In Figure 4.2 appears the definition of a generic function COMPARE-AND-SWAP that can be realised in NEBULA as a single uninterruptible machine instruction. In this definition we use a non-Ada feature. The generic type ELEM is said to be instantiatable with any scalar or access type. As this corresponds to none of the generic formal types of Ada, we have used the notation

((<>))

for it[1]. Now that we have the generic function COMPARE-AND-SWAP we are in position to implement singly linked queues as the package presented in Figures 4.3 and 4.4. In this package two ENQUEUE operations are defined. The only difference between them is that in one it is returned the indication of whether the queue was or not empty at the time that the operation is called. Similarly, there are two versions of the DEQUEUE operation. The more complex version determines if the queue was or not empty at the time that the operation was called.

The ENQUEUE and DEQUEUE operations are extremely rapid. Implemented as an inline procedure the ENQUEUE operation almost always can be completed in 5 machine instructions. DEQUEUE almost always will require also only 5 machine instructions. [The probability that two tasks execute ENQUEUE and DEQUEUE operations on the same queue at the same time is very low]

Although these operations on queues are implemented without using any explicit synchronisation mechanism, they are safe even when invoked by concurrent tasks. In the cases where the instructions of different operations become interleaved, the overall effect of the different operations remains the same as if they had been executed in strict sequence.

In Ada tasks that call concurrently an entry are scheduled in strict FIFO fashion. Instead, tasks that are in the ready queue are executed on the basis of their priorities.

[1] Ada gives a special status to scalar types and to access types in that objects of these types can be efficiently shared. However, this distinction is lost when one gets to the generic formal types.

The singly linked queues considered above are excellent for supporting FIFO scheduling, but are not directly usable for scheduling tasks on the basis of their priorities. For the case of the ready queue a multi-level priority queue is more appropriate.

The ENQUEUE operation for multilevel queues is rapid because it involves an indexing operation on an array of queues followed by a simple ENQUEUE operation. It requires 7 instructions. The DEQUEUE operation for multilevel queues is more time consuming because it must first determine the queue of highest priority that is not empty. We estimate that this operation on the average is equivalent to 5 simple DEQUEUE operations, that is, it requires 25 instructions. In the case of semaphores, as they are not directly defined by the Ada language, there are uncertainties as to the way they should be scheduled, if with a FIFO discipline, or with a priority discipline. At first sight it would seem that semaphores should have priority scheduling. But if semaphores are used exclusively to implement critical regions that are very brief, then the FIFO discipline becomes adequate. Since our use of semaphores involves critical regions that last at most a few tens of machine instructions, the FIFO discipline can be adopted and the simple implementation shown in Figures 4.5 and 4.6 can be used.

In this implementation of semaphores we have assumed the existence of the function LINK-TO-TCB which, given the address of the WAITING-LIST field of a task, determines the address of the TCB of that task. This function, in any reasonable architecture, is trivial to implement. It is hence assumed without any further discussion. With the name SELF we mean a variable of type TQ-TCB pointing to the Task Control Block of the task executing the code where SELF appears. We assume that SELF is defined in the package KERNEL-INTERFACE. It is assumed to be read-only.

If we examine the implementation of the P and V operations of semaphores, we see that the execution of a P operation involves almost always only 6 machine instructions, and that the execution of a V operation involves almost always only 11 machine instructions. Further, these operations in most cases are executed without using any privilege or any special priority. [Our only use of semaphores is to implement very brief critical regions. Hence the probability of two tasks accessing the same semaphore at the same time is low]

```

generic
  type ELEM is (<<>>); -- as discussed in the text, this is not a
                        -- legal generic type parameter in Ada.
function COMPARE_AND_SWAP(THE_NEW:      in ELEM;
                          SHARED:      in out ELEM;
                          COPY:      in out ELEM) is
  B:    BOOLEAN;
begin
  B := (SHARED = COPY);
  if B then
    SHARED := THE_NEW;
  else
    COPY := SHARED;
  end if;
  return B;
end COMPARE_AND_SWAP;

```

FIGURE 4.2

```

package QUEUE is
  ---
  type LINK;
  type LINK is access LINK;    -- This seems legal in Ada
  type ANCHOR is
    record
      FRONT, BACK: LINK := null;
    end record;
  type AT_ANCHOR is access ANCHOR;
  ---
  procedure ENQUEUE(A: in out LINK; B: in LINK);
  procedure ENQUEUE(A: in out LINK; B: in LINK;
                    NOT_EMPTY: out BOOLEAN);
  procedure DEQUEUE(A: in out LINK; B: out LINK);
  procedure DEQUEUE(A: in out LINK; B: out LINK;
                    NOT_EMPTY: out BOOLEAN);
  ---
end QUEUE;

```

FIGURE 4.3

```

with COMPARE_AND_SWAP;
package body QUEUE is
----
    function CMPS is new COMPARE_AND_SWAP(LINK);
    procedure ENQUEUE(A: in out ANCHOR; B: in LINK) is
        T:      LINK      := A.FRONT;
        L:      BOOLEAN;
    begin
        B.all := null;
        loop
            L := CMPS(B, A.FRONT, T);
            exit when L;
        end loop;
        T.all := B;
    end ENQUEUE;
----
    -- The other version of ENQUEUE is essentially the same: it just
    -- returns in the third parameter the value of L.
----
    procedure DEQUEUE(A: in out ANCHOR; B: out LINK) is
        V:      LINK;
        L:      BOOLEAN;
    begin
        B := A.BACK;
        while B /= null loop
            V := B.all;
            L := CMPS(V, A.BACK, T);
            exit when L;
        end loop;
    end DEQUEUE;
----
    -- The other version of DEQUEUE is essentially the same: it just
    -- returns in the third parameter the value of L.
end QUEUE;

```

FIGURE 4.4

```

with QUEUE;
package SEMAPHORE_PACK is
----
    type SEMAPHORE is limited private;
    procedure P(S: in out SEMAPHORE);
    procedure V(S: in out SEMAPHORE);
----
private
    type SEMAPHORE is QUEUE.ANCHOR;
end SEMAPHORE_PACK;

```

FIGURE 4.5

```

with LINK_TO_TCB, KERNEL_INTERFACE;
use QUEUE, KERNEL_INTERFACE;
package body SEMAPHORE_PACK is
---
  procedure P(S: in out SEMAPHORE) is
    NOT_EMPTY:      BOOLEAN :=      TRUE;
  begin
    ENQUEUE (S, SELF.WAITING_LIST, NOT_EMPTY);
    if NOT_EMPTY then
      GIVE_UP;
    end if;
  end P;
---
  procedure V(S: in out SEMAPHORE) is
    NOT_EMPTY:      BOOLEAN :=      TRUE;
    X:              AT_TCB;
    Y:              LINK;
  begin
    DEQUEUE(S, Y);      -- Dequeue Self
    DEQUEUE(S, Y, NOT_EMPTY); -- Dequeue a waiting task, if any
    if NOT_EMPTY then
      X := LINK_TO_TCB(Y);
      if X.CURRENT_PRIORITY > SELF.CURRENT_PRIORITY then
        GIVE_UP_WITH_SUBSTITUTE(X);
      else
        READY_ENQUEUE(X);
      end if;
    end if;
  end V;
---
end SEMAPHORE_PACK;

```

FIGURE 4.6

4.3 SOME BASIC TASK INTERACTIONS

In Figure 4.7 are the specifications of some packages and procedures that we will use in implementing some of the basic rendez-vous mechanisms of Ada. In addition we will assume available schemas for generating code in correspondence to the phrases described below.

The phrases

PREPARE_PARAMETER_LIST

and

COPY_BACK_PARAMETER_LIST

indicate respectively the actions taking place:

- when the parameters of an entry_call are readied, and
- when, after the call is completed, the 'out' and 'in out' parameters are copied back to the caller.

The PREPARE_PARAMETER_LIST and COPY_BACK_PARAMETER_LIST are performed by the task acting as caller in the rendez-vous.

The phrases

GET_PARMS

and

PUT_PARMS

indicate respectively the actions taking place:

- when the parameter list prepared by the caller is readied for the callee, and
- when this list is readied back for the caller.

The GET_PARMS and PUT_PARMS actions are performed by the task acting as callee in the rendez-vous.

What PREPARE_PARAMETER_LIST, GET_PARMS, PUT_PARMS, and COPY_BACK_PARAMETER_LIST actually are depends very much on what the parameters are as to type, number and mode. Under the assumptions we have made about address spaces, no significant concurrency issue is involved in these operations, and we will not dwell further on them.

In Figure 4.8 appears the code corresponding to the simple entry call:

```
T.e(....);
```

In Figure 4.9 appears the code corresponding to the Accept statement:

```
accept e(...) do
  -- a sequence of statements S
end e;
```

In Figure 4.10 appears the code corresponding to the Conditional Entry Call Statement:

```
select
  T.e(...);
  -- a sequence of statements S1
else
  -- a sequence of statements S2
end select;
```

In Figure 4.11 appears the code corresponding to the Timed Entry Call Statement:

```
select
  T.e(...);
  -- a sequence of statements S1
or
  delay D;
  -- a sequence of statements S2
end select;
```

In Figure 4.12 appears the code corresponding to the Selective Wait Statement without Delay Alternative, or Else Alternative, or Terminate Alternative:

```
select
  when g1 => accept e1(...) do
    -- a sequence of statements S1
    end e1;
    -- a sequence of statements S1'
or
  .....
or
  when gN => accept eN(...) do
    -- a sequence of statements SN
    end eN;
    -- a sequence of statements SN'
end select;
```

The cases of the Selective Wait Statements with Delay, or Else, or Terminate Alternative are not considered as they do not add substantially to our understanding of the rendez-vous mechanism or of its implementation.

The code we show for the cases we consider is fairly long but of only limited complexity. A substantial portion of the apparent complexity is due to the need to check on possible requests to abort the given task and to the need to propagate exceptions arising during rendez-vous in both the caller and the callee.

A Simple Entry Call that is without parameters and that finds the callee waiting requires:

- 13 instructions, and
- a P operation (at least 6 instructions)
- a V operation (at least 11 instructions)
- an ENQUEUE operation (at least 5 instructions), and
- a GIVEUPWITHSUBSTITUTE operation.

That is, some 32 instructions plus the time required to carry out the GIVE-UP-WITH-SUBSTITUTE operation. If the callee task is not waiting,

it will take 34 instructions plus a GIVE-UP operation. Similarly we can estimate that an Accept Statement that is without parameters and that finds a caller waiting requires 50 instructions. Otherwise it requires 77 instructions and a GIVE-UP operation. Hence in the best of circumstances a rendez-vous requires 84 instructions plus a GIVE-UP operation. All this code is executed without the need to turn off interrupts.

As the reader may have noticed, our implementation of the rendez-vous mechanism does not strive for efficiency, just for expository simplicity. Hence our estimates are upper bounds.

```
task DELAYER is
---
    entry DELAY(T: AT_TCB; D: DURATION);
    entry CANCEL(T: AT_TCB);
---
end DELAYER;

procedure SPREAD_ABORT(T: AT_TCB);

task TERMINATOR is
---
    entry TERMINATE(T: AT_TCB);
---
    -- When task T is terminated all area associated with T are
    -- reclaimed by the system. Of course, TERMINATE will be
    -- also applied to each of the tasks dependent on T.
---
end TERMINATOR;

package KERNEL_INTERFACE is
---
    SELF:          constant AT_TCB;
    --
    procedure GIVE_UP;
    --
    procedure GIVE_UP_WITH_SUBSTITUTE(X: AT_TCB);
    --
    procedure READY_ENQUEUE(X: AT_TCB);
    --
end KERNEL_INTERFACE;
```

FIGURE 4.7

```

if SELF.STATUS = ABORTED then
    TERMINATE(SELF);
elsif T.STATUS /= CALLABLE then
    raise TASKING_ERROR;
end if;
P(T.MUTEX);
ENQUEUE(T.WLIST(e).FRONT, SELF);
SELF.TASK-BEING_CALLED := T;
SELF.ENTRY_BEING_CALLED := e;
if not T.GATE(e) then
    V(T.MUTEX);
    PREPARE_PARAMETER_LIST;
    GIVE_UP;
else
    T.GATE := (others => FALSE);
    V(T.MUTEX);
    PREPARE_PARAMETER_LIST;
    GIVE_UP_WITH_SUBSTITUTE(T);
end if;
if SELF.STATUS = ABORTED then
    TERMINATE(SELF);
end if;
if SELF.RESULT_WAS = WITH_EXCEPTION then
    raise SELF.EXCEPTION_IS;
end if;
COPY_BACK_PARAMETERS;

```

FIGURE 4.8

Simple Entry Call: T.e(...);

```

declare
  U:                AT_TCB;
  WAS_EMPTY:        BOOLEAN := FALSE;
  PRIORITY_LIFT:    SYSTEM.PRIORITY;
  EXCEP_CODE:        EXCEPTION_CODE := EXCEPTION_CODE'LAST;
begin
  loop
    if SELF.STATUS =ABORTED then TERMINATE(SELF); end if;
    P(SELF.MUTEX);
    DEQUEUE(SELF.WLIST(e).BACK, U, WAS_EMPTY);
    exit when not WAS_EMPTY;
    SELF.GATE(e) := TRUE;
    V(SELF.MUTEX);
    GIVE_UP;
  end loop;
  if U.WITH_DELAY_ALTERNATIVE then
    U.WITH_DELAY_ALTERNATIVE := FALSE; -- TEST and SET instruction
    V(SELF.MUTEX);
    DELAYER.CANCEL(U);
  else
    V(SELF.MUTEX);
  end if;

```

FIGURE 4.9 (cont.)

Simple Accept Statement: accept e(...) do S end e;

```

SELF.PREVIOUS_PRIORITY := SELF.CURRENT_PRIORITY;
PRIORITY_LIFT := U.CURRENT_PRIORITY - SELF.CURRENT_PRIORITY;
begin
  if PRIORITY_LIFT > 0 then
    CHANGE_PRIORITY(SELF, PRIORITY_LIFT);
  end if;
  GET_PARMS;
  S
  PUT_PARMS;
  U.RESULT_IS      := SUCCESS;
exception
  others => EXCEP_CODE      := ECODE;
           U.RESULT_IS      := WITH_EXCEPTION;
           U.EXCEPTION_IS   := EXCEP_CODE;
end;
SELF.CURRENT_PRIORITY := SELF.PREVIOUS_PRIORITY;
if PRIORITY_LIFT > 0 then
  CHANGE_PRIORITY(SELF, -PRIORITY_LIFT);
  GIVE_UP_WITH_SUBSTITUTE(U);
else
  READY_ENQUEUE(U);
end if;
if SELF.STATUS = ABORTED then
  TERMINATE(SELF);
end if;
if EXCEP_CODE /= EXCEPTION_CODE'LAST then
  raise EXCEP_CODE;
end if;
end;

```

FIGURE 4.9

Simple Accept statement: accept e(...) do S end e;

```

if SELF.STATUS = ABORTED then
    TERMINATE(SELF);
elsif T.STATUS /= CALLABLE then
    raise TASKING_ERROR;
end if;
P(T.MUTEX);
if not T.GATE(e) then
    V(T.MUTEX);
    S2;
else
    ENQUEUE(T.WLIST(e).FRONT, SELF);
    T.GATE := (others => FALSE);
    T.ENTRY_BEING_CALLED := CALLED_e;
    V(T.MUTEX);
    SELF.TASK_BEING_CALLED := T;
    SELF.ENTRY_BEING_CALLED := e;
    PREPARE_PARAMETER_LIST;
    GIVE_UP_WITH_SUBSTITUTE(T);
    if SELF.STATUS = ABORTED then
        TERMINATE(SELF);
    end if;
    if SELF.RESULT_WAS = WITH_EXCEPTION then
        raise SELF.EXCEPTION_IS;
    end if;
    COPY_BACK_PARAMETERS;
    S1;
end if;

```

FIGURE 4.10
Conditional Entry Call:

```

select
    T.e(...); S1;
else
    S2;
end select;

```

```

if SEL.STATUS = ABORTED then
    TERMINATE(SELF);
elsif T.STATUS /= CALLABLE then
    raise TASKING_ERROR;
end if;
P(T.MUTEX);
ENQUEUE(T.WLIST(e).FRONT, SELF);
SELF.TASK_BEING_CALLED := T;
SELF.ENTRY_BEING_CALLED := CALLED_e;
if not T.GATE(e) then
    SELF.WITH_DELAY_ALTERNATIVE := TRUE;
    DELAYER.DELAY(SELF, D);
    V(T.MUTEX);
    GIVE_UP;
else
    T.GATE := (others => FALSE);
    T.ENTRY_BEING_CALLED := CALLED_e;
    V(T.MUTEX);
    PREPARE_PARAMETER_LIST;
    GIVE_UP_WITH_SUBSTITUTE(T);
end if;
if SELF.STATUS = ABORTED then
    TERMINATE(SELF);
end if;
if SELF.DELAY_COMPLETED then
    SELF.DELAY_COMPLETED := FALSE;
    SELF.WITH_DELAY_ALTERNATIVE := FALSE;
    S2;
else
    if SELF.RESULT_IS = WITH_EXCEPTION then
        raise SELF.EXCEPTION_IS;
    else
        COPY_BACK_PARAMETERS;
        S1;
    end if;
end if;

```

FIGURE 4.11

Timed Entry Call:

```

select
    T.e(...); S1;
else
    delay(D); S2;
end select;

```

```

declare
  U:                AT_TCB;
  WAS_EMPTY:        BOOLEAN := FALSE;
  PRIORITY_LIFT:    SYSTEM.PRIORITY;
  EXCEP_CODE:        EXCEPTION_CODE := EXCEPTION_CODE'LAST;
  THE_ALTERNATIVE:  INTEGER range 0 .. N := 0;
  PSEUDO_GATE:       array(1 .. NO_OF_ENTRIES) of BOOLEAN
                    := (others => FALSE);
  MAPPING:          array (1 .. N) of INTEGER
                    := (e1, e2, ..., eN);
  MAPPING_1:         array (1 .. NO_OF_ENTRIES) of integer
                    := -- values are 0 or the position where
                      -- called in the Select Statement

  COUNT:            INTEGER := 0;
begin
  if SELF.STATUS = ABORTED then
    TERMINATE(SELF);
  end if;
  if g1 then COUNT := COUNT + 1; PSEUDO_GATE(MAPPING(1)) := TRUE; end if;
  ..... -- similar code for g2, g3, ...,
  if gN then COUNT := COUNT + 1; PSEUDO_GATE(MAPPING(N)) := TRUE; end if;
  if COUNT = 0 then
    raise PROGRAM_ERROR;
  end if;
  P(SELF.MUTEX);
  for I in 1 .. N loop
    if PSEUDO_GATE(MAPPING(I)) then
      DEQUEUE(SELF.WLIST(I).BACK, U, WAS_EMPTY);
      if not WAS_EMPTY then
        THE_ALTERNATIVE := I;
        exit;
      end if;
    end if;
  end loop;
  if THE_ALTERNATIVE = 0 then
    SELF.GATE := PSEUDO_GATE;
    V(SELF.MUTEX);
    GIVE_UP;
    if SELF.STATUS = ABORTED then
      TERMINATE(SELF);
    end if;
    P(SELF.MUTEX);
    DEQUEUE(SELF.WLIST(SELF.THE_ENTRY_BEING_CALLED).BACK, U);
    THE_ALTERNATIVE := MAPPING_1(SELF.THE_ENTRY_BEING_CALLED);
  end if;
  if U.WITH_DELAY_ALTERNATIVE then
    U.WITH_DELAY_ALTERNATIVE := FALSE; -- TEST and SET
    V(SELF.MUTEX);
    DELAYER.CANCEL;
  else
    V(SELF.MUTEX);
  end if;

```

FIGURE 4.12 (Cont.)

```

SELF.PREVIOUS_PRIORITY := SELF.CURRENT_PRIORITY;
PRIORITY_LIFT          := U.CURRENT_PRIORITY - SELF.CURRENT_PRIORITY;
begin
  if PRIORITY_LIFT > 0 then CHANGE_PRIORITY(SELF, PRIORITY_LIFT); end if;
  case THE_ALTERNATIVE is
    when 1 => GET_PARMs_1; S1; PUT_PARMs_1;
    .....
    when N => GET_PARMs_N; SN; PUT_PARMs_N;
  end case;
  U.RESULT_IS := SUCCESS;
exception
  others => EXCEP_CODE      := ECODE;
            U.RESULT_WAS   := WITH_EXCEPTION;
            U.EXCEPTION_IS := EXCEP_CODE;
end;
SELF.CURRENT_PRIORITY := SELF.PREVIOUS_PRIORITY;
if PRIORITY_LIFT < 0 then
  CHANGE_PRIORITY(SELF, -PRIORITY_LIFT);
  GIVE_UP_WITH_SUBSTITUTE(U);
else
  READY_ENQUEUE(U);
end if;
if SELF.STATUS = ABORTED then
  TERMINATE(SELF);
end if;
if EXCEP_CODE /= EXCEPTION_CODE'LAST then raise EXCEP_CODE; end if;
case THE_ALTERNATIVE is
  when 1 => S1'
  .....
  when N => SN'
end case;
end;

```

FIGURE 4.12

Selective Wait Statement without ELSE, DELAY, or TERMINATE

```

select
  when g1 => accept e1( ... ) do S1 end e1; S1'
or
  .....
or
  when gN => accept eN( ... ) do SN end eN; SN'
end select;

```

4.4 IMPLEMENTING THE ADA KERNEL ON NEBULA

Our discussion of the concurrency kernel of Ada assumes the availability of a few kernel services. The implementation of these services on NEBULA is straightforward, as it has features that truly simplify the solution of concurrent programming problems.

Examples of NEBULA support are:

- o The treatment of Supervisor requests and of task dispatching calls as procedure calls. Consequently it is possible to use the parameter passing mechanism of procedure calls when doing these operations. This simplifies the communication of information between a task and the kernel and between any two tasks.
- o The SVC instruction allows a rapid, safe access to the supervisor space. The procedure invoked by the SVC may be privileged and so it can perform privileged instructions to act on tasks or to change the running task's priority. This procedure can also access the control blocks of all tasks if, as logical, they appear in the supervisor space.
- o The switch from one task to another is extremely rapid, just the sequence STORE-TASK, LOAD-TASK, START-TASK (possibly this sequence is preceded by an SVC and followed by a RET instruction)
- o It is possible to start a task and to raise an exception in it with a single operation.
- o It is possible to clear the context stack of a task at the same time that it invokes another task.
- o It is possible for the kernel to check with a single instruction on the legality of a pointer passed to it.

In general, the procedure mechanism of NEBULA is successful and the overall structure of the machine simplifies considerably the implementation of control structures and, in particular, the manipulation of tasks.

The memory management facilities of NEBULA, instead, leave much to be desired.

In [SDW] are given reasons for NEBULA's memory management structure. It is explained that the NEBULA architecture will be implemented in both micros and in mainframes. The smaller machines will not allow, it is stated, the use of a two-level (segments over pages) virtual to physical address mapping because of its complexity and cost. A one-level map using pages is excluded because the use of either small pages or of large pages is unsatisfactory (small pages, because they require large page tables; large pages, because they lead to inefficient utilization of physical memory).

Because of these reasons the Authors chose the organisation that we

have described in Chapter 2. This organisation uses two separate Segment Maps, one for the User mode and the other for the Supervisor mode. In a Segment Map a Segment Descriptor does not uniquely define the size of a segment, nor does it indicate exactly the addresses mapped onto that segment. In order to determine this information it is necessary to examine the Segment Descriptor that precedes the given one. For example, if we are given Segment descriptors with bounds b_0 , b_1 , b_2 , ... then each segment will hold the following virtual addresses:

Segment 0	$0 \leq \dots \leq b_0$
Segment 1	$b_0 < \dots \leq b_1$
Segment 2	$b_1 < \dots \leq b_2$
.....	

This means that if we need to increase the size of the 0th Segment by some amount d then:

- o Either we add d to the bound of the 0th segment and leave the other segment descriptors unchanged, or
- o We increase all bounds by d .

In the first case the 1st segment becomes reduced by the same amount d that the 0th segment was increased by. In the second case all segment descriptors need to be changed. In either case a non-local effect takes place in correspondence to what should be a local change. Of course, as indicated in [INT], we could insert a 'slack' segment descriptor following the descriptor of each segment that we may want to modify in size. For example, if we know that we may want to change the size of segment 0, we may place following it a segment whose descriptor forbids all accesses and has a bound of b_0' ($b_0 < b_0'$). Then all changes on the size of segment 0 that do not make it bigger than b_0' are allowed and are local. Of course this solution is not desirable because it requires extra segment descriptors.

The virtual to physical address translation in NEBULA is not simple and requires the use of associative techniques. This limits the number of segments possible even in the larger implementations of NEBULA.

Given these problems, it is hard to prefer the segmentation schema of NEBULA over more traditional segmentation methods [SIT].

As we described in Chapter 1, it is necessary to be able to map a portion of a master task address space into a dependent task address space. This is required to allow the dependent task to access directly the objects that are declared in the master and are visible to the dependent. In addition, when two tasks act as caller and callee in a rendez-vous, parameters may be sent in either or both directions.

To accommodate these requirements there is no simple mechanism in NEBULA or, for that matter, in any other architecture we know of:

1. It is possible to use totally distinct address spaces in distinct tasks. But the the access to shared objects becomes too slow because each access must be mediated by a supervisory procedure.
2. It is possible to set up a form of controlled space sharing between tasks by copying certain segment descriptors of the Segment Map of the parent task into the Segment Map of the dependent task. This requires some set up when the dependent task is activated. More serious is the fact that it requires a substantial number of segments. In addition it does not help at all with the problem of parameter passing between tasks during rendez-vous.
3. One can adopt the solution that we have been using in this Chapter. This solution uses a single virtual address space for all the tasks of a program. This solution reduces substantially the protection possible between concurrent tasks. It also requires a substantial number of segments, if we intend to use efficiently physical memory. Its only advantage is execution speed.

Of these three approaches, probably the third is the most efficient. But it uses the Memory Management of NEBULA in the way that is least appropriate for it, almost as a paged memory. In any case, the conclusion that NEBULA's Memory Management does not support conveniently Ada is warranted.

Our estimate on the number of instructions that it takes to perform the kernel operations are:

5 instructions for GIVE-UP-WITH-SUBSTITUTE

7 instructions for READY-ENQUEUE, and

30 instructions for GIVE-UP. [We need to perform a READY-DEQUEUE followed by a GIVE-UP-WITH-SUBSTITUTE]

It follows that in the best of circumstances a simple Entry Call will require 40 instructions and a simple Accept statement will require some 50 instructions. The fastest rendez-vous possible requires 114 instructions. This, as stated earlier, is in the case that no attempt is made to optimize the code. But it remains to be seen how much optimization is actually feasible.

5.0 SUMMARY

In this report we have examined how well the NEBULA architecture supports the concurrency features of Ada with the ultimate aim of determining the suitability of using the NEBULA Architecture to execute real-time Ada programs.

We have reviewed the concurrency aspects of Ada. We have scrutinized in particular the storage structure of executing Ada programs. We have found that the Ada policy of allowing task declarations within tasks, and of supporting block-structured visibility rules, substantially complicates addressing within a program. Storage management is equally complex because a Cactus Stack structure is required.

In evaluating the ease of implementing a concurrency kernel for Ada on a machine, two aspects need to be examined above all:

the facilities provided to support context switches from one task to another or from one task to the supervisor, and

the structure of the available Virtual Address Spaces.

We have reviewed the NEBULA architecture from these two points of view.

NEBULA has a number of novel features intended to simplify procedure calls and context switches. Context Stacks simplify considerably the preparation required to perform a procedure call, and procedure calls are used as the basic control structure. Supervisor calls, Interrupts, traps are all interpreted as procedure calls. The context switch from a task to another can be accomplished within a maximum of 5 instructions. A global Context Stack, the Kernel Context Stack, is available during interrupt and trap handling. At all other times system services, just like user code, use Task Context Stacks.

NEBULA supports a one level segmented Virtual Address schema. The size of a segment is variable up to the size of the whole physical memory. Two Segment Maps are used to translate from virtual to physical addresses: the User and Supervisor Maps. The User Map is associated to the individual task, the Supervisor Map is the same for all tasks. The Supervisor Map is usable only when in Supervisor mode.

We have then examined the implementation of the concurrent mechanisms of Ada in terms of its sequential features and of a basic set of operations, of a concurrency kernel. Queues and semaphores were easily implemented in terms of these operations and then used in turn in the implementation of various rendez-vous mechanisms. This study was done in a fashion independent of the underlying architecture. Only later the use of NEBULA was considered.

The control structures of NEBULA were found extremely supportive of the Ada concurrency mechanisms [NEBULA would be equally supportive of other concurrent programming languages or, for that matter, of machine language real-time programming].

Instead no truly satisfactory way was found to utilize the Virtual Address spaces of NEBULA to support the controlled memory sharing

policy of Ada.

The execution of a Simple Entry call was found to require 40 instructions when immediately successful. The execution of an Accept statement, when immediately successful, required 50 instructions. The minimum cumulative (caller + callee) number of instructions required by a rendez-vous is 114 instructions. These figures were obtained without any attempt at optimisation. But it seems unlikely that any local optimization will substantially improve on this figure.

Various features of NEBULA that do not appear in other architectures were helpful. In the implementation of these basic rendez-vous mechanisms NEBULA saves some instructions with respect to the VAX because of its more convenient task oriented instructions.

To know that the simplest rendez-vous mechanism requires 40 instructions is a useful piece of information. But of itself does not help determine the usability of Ada to implement real-time systems on NEBULA (or, for that matter, on any other machine). Though the research being reported did not consider in sufficient depth this question, a number of observations follow from our study:

1. Our estimates on the number of instructions required to carry out rendez-vous mechanisms depend very little on NEBULA. NEBULA is more helpful in this implementation than other architectures. The complexity of the general Ada rendez-vous mechanism is at the root of the problem.
2. One of the aims of Ada was to allow the use of concurrent tasks in all the situations where it would be conceptually advantageous to do so. It appears that this will result in a substantial penalty in the efficiency of the resulting program.
3. If a semaphore is implemented directly in NEBULA assembly language, the execution of a P operation requires the execution of 6 instructions. If the same semaphore is implemented in Ada, the execution of a P operation requires the execution of over 100 instructions.

It would appear that:

either compilers must be developed that are capable of pretty fancy optimizations, or

in applications where the real time constraints are particularly stringent the careful use of packages written in low level language will still be required.

BIBLIOGRAPHY

- [ADA] Department of Defence: Reference Manual for the
Ada* Programming Language
ANSI/MIL-STD-1815 A January 1983
- [Dietz, Szeuwerenko: Architectural Efficiency Measures:
An Overview of three Studies
Computer, April 1979, 26-33
- [INT] Intermetrics: NEBULA as a Target for Ada
Report to Battelle Columbus Laboratories
Delivery Order No. 1698
- [DVH] Dennis, VanHorn: Programmed Semantics for
Multiprogrammed computations
Comm. of ACM, 3:143-155 (1966)
- [NEB] MIL-STD-1862B Military Standard:
NEBULA Instruction Set Architecture
- [SDW] Szeuwerenko, Dietz, Ward: NEBULA, a New Architecture and its
Relationship to Computer hardware
Computer, February 1981, 35-41
- [SIT] Sites, Richard: Operating Systems and Computer Architecture
in: Introduction to Computer Architecture
(H. S. Stone) SRA, 1980
- [TAN] Tanenbaum, A.: Implications of Structured Programming
for Machine Architecture
Comm. of ACM, 21:237-246 (1978)

END

DATE
FILMED

10 — 83

DTIC