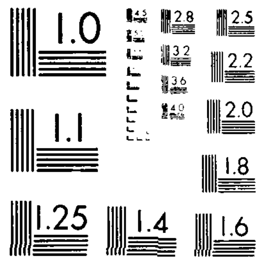
AD-A193 051 ADA TASKING IN TEN15(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN (ENGLAND) D J TOMBS OCT 87 RSRE-MEMO-4074 DRIC-BR-104331 1/1

UNCLASSIFIED F/G 12/5 NL

END
DATE
FILMED
7 8-



MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

RSRE

MEMORANDUM No. 4074

ROYAL SIGNALS & RADAR ESTABLISHMENT

AD-A193 051

DTIC
ELECTE
MAY 1 2 1988
S D

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4074

Title ADA TASKING IN TEN15

Author D J TOMBS

Date OCTOBER 1987

Summary

An Ada compiler developed at RSRE uses the Ten15 abstract machine as its target. This paper describes in detail the Ten15 implementation of the tasking support for this compiler.

A-1

Copyright
©
Controller HMSO London
1987

Contents

1 Introduction

2 Concurrency in Ten15

3 A Summary of Tasking in Ada

4 Ada Tasking in Ten15 - Overall Structure
- 4.1 Interface with Compiler
- 4.2 Internal Organisation

5 Ada Tasking in Ten15 - Details
- 5.1 Declaring a Task
- 5.2 Activation and Termination
- 5.3 Rendezvous
- 5.4 Abortion
- 5.5 Attributes and Pragmas

6 Conclusion

7 References

8 Appendices
- A Ten15 Concurrency Operators
- B A Task in Ten15
- C Library Suite Routines
- D An Example Task Declaration

1 Introduction

Ten15 is an algebraic abstract machine which can also be considered as a structured, strongly-typed code used to develop programs. It is used by high-level language compilers as an intermediate target independent of language and hardware. The abstraction of Ten15 means that compiled code is better defined and at a higher level than that output for a conventional operating system.

Currently an Ada † compiler for the Ten15 abstract machine is under development [4],[5]. The objective of this paper is to describe the Ten15 implementation of tasking for this compiler.

Chapter two discusses the primitives in Ten15 which represent parallel processes and synchronisation.

The Ada language specifies a form of parallel processing known as tasking. A brief overview of tasking is given in chapter three as an introduction to the terminology.

Conventional operating systems, eg [6], perform tasking through a set of machine-level routines, commonly known as the run-time library, which run the Ada program and organise time sharing between tasks. With Ten15 it is the underlying hardware implementation which maintains process scheduling, for all programs and independent of language, hence Ada tasking is considered only in terms of Ten15 processes. Each Ada operation is performed either directly as compiled Ten15 or by accessing a procedure from a Ten15 library suite which is incorporated into the compiled code as something non-local to the entire program. The implementation of the Ten15 and a description of the suite form the bulk of this paper, in chapters four and five.

† Ada is a registered trade mark of US DoD Ada Joint Program Office.

2 Concurrency in Ten15

Ten15 is an abstract machine defined algebraically. A program in Ten15 is considered as a state transformation within the Ten15 algebra. The algebraic content provides a solid base for program transformation and analysis and is rich enough to provide a usable interface to the compiler-writer or programmer. A survey of the algebra, its constructions, states and values is given by [2].

Each value in the Ten15 machine is strongly typed, the associated type determining the possible operations on the value. The range of Ten15 types is sufficient to describe the computer universe. It includes types appropriate to objects in a high level language: integers, strings, variables and the like, and also to such concepts as persistent filestore or processes. In particular, Ten15 procedures are first class values, described in terms of closures, with every runnable Ten15 program closely bound to its environment.

Constructions in the Ten15 algebra are classified into various sorts. Those corresponding to high-level language control statements are of one sort, branches of another, and operators which produce new values from existing (typed) values are a third sort. Within this structure the concurrency aspects of Ten15 are defined as a small number of typed values and operators which act on them.

Concurrency in Ten15 is considered in terms of processes - essentially procedures applied in parallel - with synchronisation enabled through binary semaphores ("barriers"). Processes and barriers are generated and manipulated procedurally through Ten15 operators. There is one operator, **LaunchProcess**, relating to processes, and also one **NewBarrier** to create a barrier. These and other Ten15 concurrency operators are detailed in Appendix A.

3 A Summary of Tasking in Ada

The following is a brief summary of the tasking aspects of Ada. For a full exposition the reader is referred to the Language Reference Manual [1]. The summary introduces terminology used by the rest of the paper. The description of the implementation illustrates some of the more complex details such as the rules governing exceptions and termination.

A task is an Ada unit which runs in concurrency with other tasks. It is declared with specification and body separate, like a package, and is similarly identified by name. The only objects kept in the specification and visible to the outside world are entries, which are the means of synchronising and communicating between tasks. An entry has similarities with a procedure: it has parameters of various types and is called in a similar manner. Different entries with the same parameter types can be kept as an array or family. As well as individual task objects a task type can be declared, also as a pair (specification, body). Thereafter different tasks with the same specification and body can be declared using the type definition, and arrays of tasks, access-type tasks, etc. can also be generated.

A task, procedure or block is a master of subsidiary tasks created during its execution. A task depends on its immediate master and also indirectly on any unit the master depends on. The exception to this rule is a task which has an access type, which is directly dependent upon the master that elaborated the corresponding type definition. Thus the set of running tasks within the Ada program, together with any procedures or blocks used to create them, form a dependency tree. This tree extends to cover tasks declared in separately compiled units.

A task is launched or activated following the successful elaboration of its master. After activation the two tasks then continue in parallel.

The importance of the dependency tree is in the handling of termination. When a task, procedure or block reaches its **end** it is said to be completed. A completed task, procedure or block terminates as soon as all dependent tasks have terminated.

Synchronisation and communication is enacted through a rendezvous on one of the entries of some task. The two sides of a rendezvous are asymmetric, the task owning the entry accepting a call of the entry; any task may call the entry; either task will wait for the other to synchronise. The calling task is placed on a queue for the called entry. A timeout can be attached which will cancel the call. An accept

statement for an entry either stands alone or forms a limb of a select statement. A task accepting an entry in a simple, non-selective manner engages in a rendezvous with the call at the head of the entry queue; if the queue is empty, the accepting task is suspended. A selective accept can offer a choice between more than one accept, and in addition can offer one of the following: a delay alternative or an else part which give a timeout in the same manner as an entry call; or an alternative which will terminate the accepting task when all other tasks which could communicate with it have themselves terminated or are suspended making such a selection. When the rendezvous commences the acceptor may optionally obey a sequence of statements (the "do" part), during which period the caller is suspended. When this has finished both tasks continue, asynchronously, in parallel.

A task can be explicitly terminated using an abort statement. The effect of this is to mark the task being aborted and all of its dependants as abnormal, and then to terminate them all before any communication with any other task.

4 Ada Tasking - Overall Structure

This chapter provides a broad overview of the structure of the tasking support and its interface with the Ada compiler. Chapter five is a detailed description of support software.

4.1 Interface with Compiler

The tasking aspects of the Ada compiler in part generate their own code and in part access a set of preprepared Ten15 procedures - the tasking suite.

Much of the code to implement Ada tasking is contained in a library suite. This suite is written in Ten15 assembler notation, a high-level language embodying the "program-code" nature of Ten15 which facilitates the generation of any desired Ten15 [3]. Wherever this paper uses examples of the notation a limited explanation is given. The compiler incorporates required routines into the output code as some constant of the entire compiled Ada program and thereafter calls on them as necessary.

Wherever possible an individual suite routine represents a specific part of Ada syntax, an accept statement for example. To simplify the compilation process the suite handles pieces of tasking that are potentially awkward to compile (task specification and body declared in separate units, long select statements, etc) as and when they occur in the Ada source text. There are other tasking constructs where the complete code is generated, more efficiently, "in-line".

4.2 Internal Organisation

Each Ada task runs in a separate Ten15 process of its own. Its current state is represented internally by a data structure with Ten15 type *Task*, described below. There is a 1:1:1 correspondence between an object with this type : a Ten15 process : an Ada task.

The main Ada program also runs in a separate process and has a *Task* data structure; the user process which calls the program is suspended until program completion and is otherwise used only to service external break-in. Thus an executable Ada program resembles any other Ada task.

The type *Task* is a pointer to a structure of fields, each of which describes some aspect of the state of the task. For further detail see Appendix B.

Control of access to an object of type *Task* during critical pieces of code and maintenance of consistency when synchronising between asynchronous processes, eg starting a rendezvous, is kept by mutual

exclusion. To do this a Ten15 barrier within each *Task* can be secured and released.

To model the network of task dependencies within the Ada the type has a recursive part by which all live task objects in the program are linked in a tree-like manner. Each *Task* has pointers to its master and all of its immediate dependants.

There is a field representing the "state" of the task: whether it is currently active (normal running), completed or terminated (the Ada meaning), abnormal (between an abortion being signalled and carried out), committed (a rendezvous with another task will commence straightaway, both sides being committed in a single, unitary action), or terminable (in a select statement with a terminate alternative). There are two cases of this latter state, "hard" and "soft". A task is in state "hard-terminable" if and only if all of its non-terminated dependants are likewise in state hard-terminable.

The associated Ten15 process is available for failure, inspection or diagnosis. If required to wait, such as during rendezvous or termination, the process queues at a barrier particular to the *Task*, the PQ.

One component of the type *Task* is a vector representing the set of all entry families (itself a vector of entries) or single entries of the task; entries are thus indexed by pairs of integers, to indicate which family (or single entry) and which entry of the family. Each individual entry is described by a dynamic list of tasks attempting to rendezvous with it. The limbs of a select statement are represented by a list of entries, formed dynamically and not part of the type *Task*. The selected entry which is actually called is indicated by an index field in each *Task*.

5 Ada Tasking - Details

5.1 Declaring a task

An Ada task comprises three Ten15 objects:
(i) a *Task* data structure.
(ii) variables required to pass entry parameter values.
(iii) a procedure which elaborates any local declarations of the task and obeys its statements.

To create such a set of objects in Ten15 the compiled code will:

(i) Generate a *Task* data structure, by calling **create_task**.

create_task: (Task, vec Int) -> Task

The parameters are the master of the task being created, and the number of entries in each family in the form of a vector. The created task will be placed in the tasking tree by adding to the dependants of the master.

(ii) Create variables as channels to pass entry parameters, one for each distinct entry parameter type. These are just variable declarations of the appropriate Ten15 type.

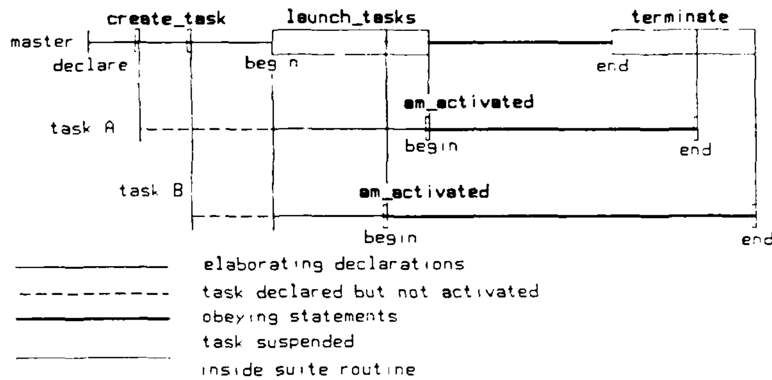
(iii) Create a procedure representing the body of the task. This will have type Task -> void. It will be called during activation with parameter the running task which launches it. Its non-locals will include the *Task* and variable just created, which are needed within the body, eg to perform an accept statement.

The above process declares an individual task object. Different task objects with the same type comprise distinct (*Task*, variable) pairs. An object with an access task type is a reference to such a pair, assigned at allocation. (Note that the master task here is that which elaborated the access type definition.) The body corresponding to any defined task type is a procedure with parameter the (*Task*, variable) pair of an object with that type which delivers the Task -> void procedure.

Declaring the outermost 'task' which is the entire Ada program is a similar process. To initialise the tasking tree **create_outer** is used instead of **create_task**, and there are no entry variables. The body procedure has the same specification as before.

5.2 Activation and termination

The activation and termination of a pair of tasks A and B within some master block is illustrated below.



Tasks are activated in 'flights' immediately after the **begin** of their master block, or if declared as part of an access-type object immediately after the object has been allocated. The launching task calls routine **launch_tasks** to activate them and suspends itself until they are all fully active, ie their declarations elaborated. **launch_tasks** is specified as:

```
launch_tasks: struct (Task, vec struct(Task, Task->void)) -> void
```

The two parameters are the launching task, and a vector comprising the set of tasks to be launched, with their corresponding bodies.

launch_tasks surrounds the body procedure of each task being launched with an algorithm for trapping and propagating exceptions and controlling termination, and applies this new procedure to the *Task* performing the launch inside a new Ten15 process. The launching task waits, on its own PQ, until all of the tasks have been successfully activated, whereupon its execution continues normally, unless at least one of the activations fails, whereupon the launching task itself is failed with Ada exception TASKING_ERROR.

When a newly-launched task has completed its activation, its first act, before any statements are evaluated, is to signal back to the launcher that the activation was successful by calling **am_activated** on two parameters: the task itself and the launching task. A failed activation will be propagated by the associated termination algorithm.

The means of terminating each task is built into it by its launch routine. After a task has completed its execution, unexceptionally or

otherwise, the Ten15 process which runs it calls procedure **terminate**. This procedure fails any tasks waiting to call an entry of the terminating task and suspends the process at its PQ until the necessary termination conditions are satisfied, ie all dependent tasks have either terminated or are simultaneously waiting at terminate alternatives inside a select statement. The task is then terminated: the running process ends and if the termination was unexceptional the *Task* data structure is removed from the tree, otherwise it is preserved to give diagnostic information. **terminate** also handles external system interrupts, eg user break-in, which fail the executing Ada program and propagates such failures to all dependants.

An Ada block, statement or procedure which directly owns a task also waits for termination after completion. The Ten15 at this point calls **terminate** explicitly, immediately after the completion of the block or procedure (having trapped any internal exceptions).

The main program simply runs within its launching procedure **launch_outer**: `struct (Task, Task->void) -> void` applied to the *Task* and body corresponding to the main Ada program. Like other tasks it runs in its own Ten15 process.

An example of an Ada task declaration, activation, and termination, and the corresponding Ten15, written in Ten15 notation, is given in Appendix D.

5.3 The Rendezvous

An Ada rendezvous is a synchronisation between two different task processes, one calling a task entry, the other accepting it.

Whichever side of the rendezvous is first encountered, synchronisation is achieved by the corresponding routine suspending the Ten15 process on its process queue, the suspension being released after its partner has reached the rendezvous and committed both sides to proceed. In order to prevent other processes accessing the tasks during these critical pieces of code, both sides first acquire their own mutual exclusion and if necessary that of their partner, and only release them at well defined points during the course of the rendezvous. With commitment made the accepting task then performs any actions necessary during rendezvous while the caller is suspended, the two tasks proceeding thereafter asynchronously.

There are three different ways of calling an entry, depending on whether the entry call is simple, conditional or timed, and also two types of acceptor according to whether the accept statement stands alone or is enclosed within a select statement alongside other accept statements, and possibly a "delay", "else", or "terminate" part also.

The interactions between processes during rendezvous are performed within procedures in the tasking suite, except that the passing and receiving of entry parameters is done within the compiled code directly.

Callers

An entry call in Ten15 is enacted in three parts: a "pre-call" to the suite routine appropriate for the type of call, which finds the correct entry and if necessary waits for it to be accepted; assignment of parameter values to the variable declared explictly for that purpose; and a "do-call" to a routine which sets the rendezvous running and waits for it to complete.

There are three types of pre-caller, to be used for entry calls which are simple, conditional, or timeout, and one universal do-caller.

Simple entry calls

simple_call: struct(Task, Task, Index) -> void

This procedure is the pre-call for making simple calls of entries of tasks. The parameters are:

(i) The *Task* corresponding to the Ada task making the call.
(ii) The *Task* corresponding to the Ada task being called.

(iii) The index of the entry being called.

simple_call checks if the called task is waiting to accept the called entry. If so it commits both sides to rendezvous, indicates which entry is being called and exits. If not, it queues the calling task on the queue for this entry, and the calling process waits on its PQ, still within **simple_call**, to exit when released. The procedure will fail with exception TASKING_ERROR if the called task is already completed or abnormal, or becomes so whilst the call is queued.

Conditional entry calls

else_call: struct(Task, Task, Index) -> bool

This procedure is the pre-call for making conditional ("select .. else") calls of entries of tasks.
Parameters are as for **simple_call**.

If the called task is waiting to accept the called entry **else_call** behaves like **simple_call**, delivering TRUE, otherwise it simply exits with FALSE. Exceptional results are as with **simple_call**.

Timed entry calls

timed_call: struct(Task, Task, Index, Duration) -> bool

This procedure is the pre-call for making timed ("select .. or delay") calls of entries of tasks. Parameters are as above, with the addition of:
(iv) the duration of the timeout.

If the called task is waiting to accept the called entry **timed_call** again behaves like **simple_call** and delivers TRUE, otherwise the call is queued. If it is subsequently released (and committed) through the entry being accepted TRUE is delivered. If the specified timeout elapses before acceptance then **timed_call** exits with FALSE. Exceptional results are as with **simple_call**.

Assignment of entry parameters

If the pre-call indicates that the rendezvous is to go ahead (that is, a commitment has been made) the values of any parameters are simply copied into the variable created for the entry in question.

Completing the call

do_call: struct(Task, Task) -> void

This procedure actually makes calls of entries. It takes two parameters:
(i) The *Task* corresponding to the Ada task making the call.
(ii) The *Task* corresponding to the Ada task being called.

do_call signals the acceptor, by now committed but suspended, to

proceed with the rendezvous and itself waits on its PQ until completion. If the rendezvous is successful a void result is delivered. Exceptions during the rendezvous are propagated to the caller. The calling task is protected from abortion while the rendezvous is in progress.

Acceptors

There are two different types of accept statement; simple and selective. A simple accept is performed by applying a single suite routine. For a selective accept, all of the alternatives in the statement (accept, delay or terminate) are individually selected beforehand, and then a routine called which makes the selection and performs the accept.

The "do" part of the Ada accept statement is implemented as a Ten15 procedure with type void->void. The entry variable is bound in by the compiler as a non-local and is assigned to by the caller. If this procedure fails internally any unhandled exception is propagated to both acceptor and caller. If the accepting task is aborted during rendezvous, TASKING_ERROR is propagated to the caller before the abortion occurs.

Simple accepts

simple_accept: struct(Task, Index, choice(void->void)) -> void

This procedure performs a simple accept of an entry of a task. The parameters of this procedure are:

(i) The *Task* making the accept.

(ii) The index of the entry to be accepted.

(iii) The action to be taken on acceptance. This is a choice between a procedure void->void to be obeyed if the accept has a "do" part, or void if there is nothing to be done.

simple_accept checks for any tasks queued calling the indexed entry. If there are none found that entry is marked "waiting to rendezvous" and the accepting process waits on its PQ, else the acceptor commits to rendezvous both itself and the task at the head of the queue, whose suspension is released. In each case the accepting task then waits until released (by **do_call**) whereupon it obeys the rendezvous routine (if any) and releases the caller, the two tasks proceeding thereafter asynchronously.

Selective accepts

Before performing a selection the various limbs of the select statement are dynamically formed into a list. Guarded limbs are included in the list only when the guard allows. There are three procedures to assist with the selection, and also a set-up to initiate the list:

start_select: void -> Selection
Initialises a selection list.

add_accept: struct(Selection, Index, Int, choice(void->void)) -> void
This procedure is called to add each accept alternative to the selection list.
The parameters are:
(i) The selection list concerned.
(ii) The index of the entry to be accepted.
(iii) An integer identifying the limb of select statement.
(iv) The rendezvous action to be taken on performing this accept (its "do" part).

add_delay: struct(Selection, Int, Duration) -> void
This procedure is called when the select statement has a delay alternative or an else part. The parameters are:
(i) The selection list concerned.
(ii) An integer identifying the limb of select statement.
(iii) The duration of the delay. For an else part this is zero. Only the delay alternative with the smallest duration is retained in the selection.

add_terminate: Selection -> void
This procedure is called when there is a terminate alternative in the select statement. The sole parameter is the selection list.

Having included all desired entries the actual selection and accept is performed by a single procedure:

select_accept: struct(Selection,Task) -> Int
The parameters are:
(i) The selection list that has just been set up.
(ii) The task performing the select.
The result of the procedure is that integer which identifies the limb of the statement selected.

select_accept checks all entries in the selection. If any have a calling task queued that which has been waiting longest is selected. Both it and the accepting task are committed to rendezvous and the caller is released from suspension. The rendezvous then proceeds as with the simple case, eventually delivering the chosen limb. What happens if no such entry is found depends on the contents of selection list:
(i) If neither a delay alternative, an else párt nor a terminate alternative are present, all entries in the selection are marked "waiting to accept" and the accepting process is suspended on its PQ. The first caller to arrive at one of the waiting entries resets all of the others, commits both sides to rendezvous and identifies itself via the "index" field of the accepting task. When released from its

PQ the acceptor obeys the selected rendezvous routine before releasing the indexed caller.

(ii) If a delay of duration zero is present **select_accept** exits with the limb identifier for that alternative.

(iii) If a delay of duration greater than zero is present all entries in the selection are marked "waiting to accept" and the accepting process is suspended. If no call of any of these arrives before the specified delay has elapsed the entries are reset not waiting and the procedure exits. Otherwise the first caller resets the entries, commits both sides, and the rendezvous proceeds as above.

(iv) In the presence of a terminate alternative **select_accept** behaves as in case (i), waiting on the task's PQ, but in one of the "terminable" states. If some master of the accepting task is then, or subsequently becomes, completed, and the conditions necessary for it to terminate are satisfied, a termination routine for the accepting task is entered which breaks its process off the PQ and fails it, the acceptor thereby terminating naturally.

Some Examples

Suppose INNER is a task whose Ada specification is

```
task INNER is
  entry E (1..3) (I: in out Integer);
  entry F (X,Y: in Float);
end INNER;
```

that is, it has a family of three entries E which take an integer value and a single entry F which takes two floating-point values. Further suppose that the variables for these entries have been declared with names inner_e_var and inner_f_var.

Then a conditional entry call of entry F

Example 1 Ada

```
select INNER.F(1.0, 2.5);
       action1;
else action2;
end select;
```

of INNER from another task OUTER would be compiled as

Example 1 Ten15

```
If else_call(outer, inner, (2,1))
Then inner_f_var := (1.0, 2.5);
     do_call(outer, inner);
     action1
Else action2
Fi
```

A simple accept statement of entry E(2)

Example 2
Ada

```
accept E(2) (I :in out Integer)
do action(I);
end accept;
```

within the body of INNER compiles into the Ten15

Example 2
Ten15

```
simple_accept( inner, (1,2)
               ToChoice Use (inner_e_var, ..)
                        In Proc action = () -> void:
                           ..
                        Enduse
             )
```

The Ten15 operator "ToChoice" applied to a void->void procedure gives a Ten15 object of type choice(void->void).

A select statement which offers a choice between a guarded accept of entry E(3), an accept of entry F and a delay alternative

Example 3
Ada

```
select
   when cond =>
      accept E(3) (I :in out Integer);
      action3;
or accept F(X,Y :in Float);
   do action(X,Y);
   end accept;
or delay 2.0;
   action4
end select;
```

becomes

Example 3
Ten15

```
Let selection = start_select()
In If cond
   Then add_accept(selection, (1,3), 1, Null{});
   Fi;
   add_accept( selection, (2,1), 2,
               ToChoice Use (inner_f_var, ..)
                        In Proc action = () -> void:
                           ..
                        Enduse
             )
   add_delay (selection, 3, Duration 2.0);
   Case select_accept (selection, inner)
   In 1 Then action3
   Elin 2 Then ()
   Elin 3 Then action4
   Esac
Ni
```

Consider what happens if the conditional entry call in example 1 attempts to rendezvous with the selective accept in example 3. Barring abortions and exceptions there are three possibilities.

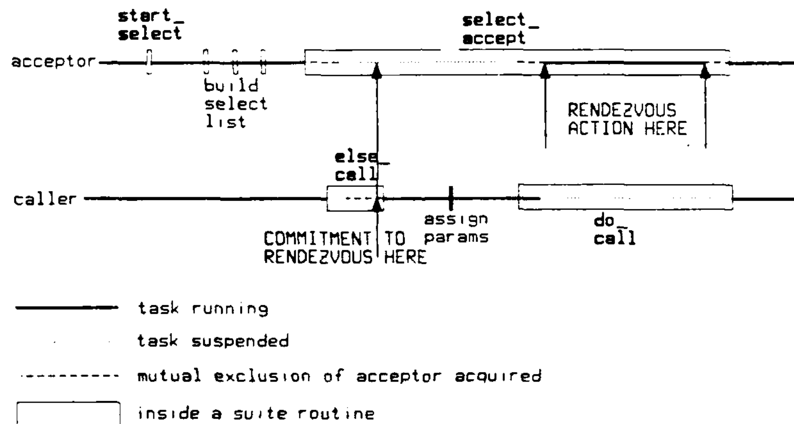
(i) The entry is called first.
(ii) The accept arrives first and no call arrives before time out.
(iii) The accept arrives first and is called before time out.

In case (i) **else_call** sees that the indexed entry has not been set waiting to receive a call and simply exits, delivering False, the caller then performing action2.

In case (ii) **select_accept** checks the entries allowed by the selection (*those indexed by (2,1) and conditionally (1,3)) for a queued calling* task. No such task is found, therefore these entries are set waiting and the accepting process waits. Nothing further happens before the timeout elapses, when the wait ends. **select_accept** delivers integer 3, whereupon action4 is performed by the case statement.

In case (iii) **select_accept** waits as before. Now **else_call** is entered before timeout, queueing the calling task on the indexed entry (2,1). This entry is waiting for a call, so **else_call** commits both tasks to rendezvous and delivers True, the parameter variable is assigned, and **do_call** entered. **do_call** lifts the wait on the acceptor and itself waits until end of rendezvous. On the accept side the rendezvous procedure action is called. When this has finished **select_accept** releases the caller, which performs action1 asynchronously, and finally delivers integer 2, causing action3 to be performed.

The diagram below illustrates case (iii). It has been drawn with the calling task arriving momentarily after the acceptor, to be suspended briefly waiting for its mutual exclusion.



5.4 Abortion

The abortion of a task is performed by calling suite routine **abort**. This takes two parameters, the *Task* performing the abortion and vector of *Tasks* to be aborted. The processes running each task in the vector will be failed asynchronously (that is, not necessarily waiting until an Ada synchronisation point is reached) with exception ADA_ABORT, as will every dependant of such a task. Should a task attempt to abort itself that operation is executed only as the last action of procedure **abort**. As with rendezvous, abortion of a task takes place under the protection of mutual exclusion to control its occurrence around critical areas.

If a task is aborted whilst calling an entry during rendezvous the abortion is suspended until the rendezvous is finished, after which the called task continues normally. If it is accepting a call during rendezvous, Ada exception TASKING_ERROR is passed to the caller before the abortion occurs.

5.5 Attributes and Pragmas

Tasking attributes T'CALLABLE and T'TERMINATED are performed by inspecting the state of the task to determine whether its entries can be called or whether it has terminated. E'COUNT is implemented as routine **e_count** which takes a task and an index, delivering the length of the queue as an integer. These actions are asynchronous and unprotected by mutual exclusion.

Pragma PRIORITY is not implemented since all processes in Ten15 are time-shared with equal priority. The conditions demanded by pragma SHARED apply by default: every update of every Ten15 reference is necessarily a unitary action.

6 Conclusion

The Ten15 tasking support suite described in this paper exists and is working. Basic checks have been made to the software using a test harness which simulates Ada tasking through Ten15 notation.

The existing partially-developed Ten15 Ada compiler will support tasking in the near future. The compilation algorithm for Ada tasking syntax has been designed and little difficulty is expected incorporating it into the compiler. Some aspects of tasking related with separate compilation will need further work owing to the incompleteness of the current compiler.

References

[1] "Reference Manual for the Ada Programming Language," US DoD Ada Joint Program Office, January 1983.

[2] "Ten15 - An Overview," Foster J.M. & Core P.W., RSRE Memorandum 3977, 1987.

[3] "A Notation for Ten15," Goodenough K.H., RSRE CC2 Divisional Memorandum, in preparation.

[4] "Ten15 and the RSRE Flex Ada Compiler," Goodenough S.J., RSRE Memorandum 4094, 1987.

[5] "Ada in a Strongly-Typed Environment," Goodenough S.J., RSRE Memorandum 4095, 1987.

[6] "VAX Ada Programmer's Run-Time Reference Manual," Digital Equipment Corporation, February 1985.

Appendix A: Ten15 Concurrency Operators

Processes

LaunchProcess: struct(Param->Result, Param) -> ProcessResult
where ProcessResult = struct(trap -> void, void -> union(Result,trap,void))

This operator acts on a procedure and its parameters. It launches the procedure applied to its parameters as a separate process. A structure of two procedures is delivered as a result of launching a process. These are:

fail_process: trap -> void

which may be used to force a failure on the launched process with the given trap value, and

get_result: void -> union(Result, trap, void)

which can be called at any time, to deliver either the Result of the procedure if it has completed successfully, the trap value if it has failed, else void if it is still running.

Barriers

NewBarrier: *bool -> Barrier*
where Barrier = struct(three procedures)

The boolean parameter is the initial state of the barrier. Three procedures are delivered which manipulate the barrier:

Pass: void -> void

If when called the barrier is *TRUE* this procedure is obeyed immediately and the barrier set *FALSE*, else the calling process is suspended on a queue.

Timeout: Duration -> bool

Same as **Pass** except that if the calling process has not been released after elapse of the duration given by the parameter it is removed from the queue and allowed to continue. This gives result TRUE, otherwise FALSE.

Raise: void -> void

If a process is queued when this is called then the one at the front of the queue is removed from the queue and released, else the barrier is set *TRUE*.

Timed wait

Delay: Duration -> void

Suspends the calling process for an amount of time given by the parameter.

Failure protection

MakeUnfailable: (Param -> Result) -> (Param -> Result)

PermitFail: (Param -> Result, Param) -> union(Result, trap)

It is sometimes necessary to protect critical sections of a process against external failure. **MakeUnfailable** produces a new procedure from an existing one which delays failure (from **fail_process**) of the

process calling it until the procedure is exited. PermitFail applies a procedure to its parameters with this protection released, trapping any resulting failures.

Appendix B: A Task in Ten15

Below is a definition of the Ten15 mode as it might appear in Ten15 notation, with "field selectors" in comments.

```
Modes:
State = 0..6;
Index = (0..Maxint,0..Maxint);
Int = Minint..Maxint;
PV = void->void;

Duration = 0..Maxint;
Doproc = choice(PV);
cycle ( AQ = ptr struct (AcceptQ, Int, Doproc, Index);
        AcceptQ = choice AQ
      );
Delay = struct(Int, Duration);
Selection = struct(AcceptQ, Delay);

Barrier = struct(PV, Duration->bool, PV);
Proc2 = union(void, trap, void);
Process = struct(trap -> void, void -> Proc2);
cycle ( Task = ptr struct( vec Family,  \Entries
                           Index,       \Index
                           State,       \State
                           PV,          \Secure
                           PV,          \Release
                           Barrier,     \PQ
                           Process,     \ProcessResult
                           Task,        \Master
                           TaskQ        \Dependants
                         );
        TQ = ptr struct (TaskQ, Task);
        TaskQ = choice TQ;
        Entry = struct( bool,           \Waiting
                        TaskQ           \EQ
                      );
        Family = vec Entry
      )

Finish
```

The fields of the structure of a *Task*, T say, are as follows:

Entries: a vector, one for each entry or family of entries of T.

Index: during a selective accept, the index of the entry being called.

State: the current state of T.

Secure and **Release** a barrier for the process running T to implement mutual exclusion.

PQ: the "process queue". Though formally a queue only the one process can wait here.

ProcessResult: the result delivered when the process running T is activated. It comprises one Ten15 procedure to fail the process and one to test for completion and give any result.

Master: the master of T; this is the data-structure corresponding to the (unique) Ada task that T is most directly dependent upon. If there is no such task its master is the main program; the main program has a null master. This field is necessary during termination and terminate accepts.

Dependants: the converse to the above: a list of tasks which have T as their direct master. Is used in termination, abortion, etc.

Each Entry E in the set of of a Task T is a pointer to a structure describing a single entry of T. E contains the following fields:

Waiting: an accept of E is waiting for a call to arrive.

EQ: a list of calling Tasks awaiting an accept of E. Thus there cannot be an item in this list when **Waiting** is True.

The limbs of a select statement are described by a Selection comprising two parts, representing the selected entries and the timeout respectively. The entries accepted in the selection form a list, each cell of which contains a "do" part, indexes the entry being accepted, and identifies the corresponding select limb. A limb identifier is also associated with the permitted timeout.

Appendix C: Library Suite Routines

The following is a list of all routines in the support suite. Their functions are described in the main body of the paper.

Creating tasks

create_task: struct(Task, vec Int) -> Task
create_outer: void -> Task

Activation and termination

launch_tasks: struct (Task, vec struct(Task, Task->void)) -> void
launch_outer: struct (Task, struct(Task, Task->void)) -> void
am_activated: struct (Task, Task) -> void
terminate: struct (Task, Task, trap) -> void

Calling entries

simple_call: struct (Task, Task, Index) -> void
else_call: struct (Task, Task, Index) -> bool
timed_call: struct (Task, Task, Index, Duration) -> bool
do_call: struct (Task, Task) -> void

Accepting calls

simple_accept: struct (Task, Index, choice(void->void)) -> void
start_select: void -> Selection
add_accept: struct(Selection, Index, Int, choice(void->void)) -> void
add_delay: struct(Selection, Int, Duration) -> void
add_terminate: Selection -> void
select_accept: struct(Selection,Task) -> Int

Abortion

abort: struct(Task, vec Task) -> void

Attributes

e_count: struct(Task, Index) -> Int

Appendix D: An Example Task Declaration

This example is of an Ada task declaration, activation, and termination, and the corresponding Ten15, written in Ten15 notation.

Ada

```
procedure OUTER is

  task INNER is
    entry E (I : in out Integer);
  end;

  task body INNER is
    ..
    ..
  begin -- signal OUTER to continue
    ..
    ..
  end INNER;

begin -- activation of INNER
  ..
  ..
end OUTER;
```

Ten15

```
Let outer :Task = create_outer()
                  \declare task object for outer
Let outerbody = \declare procedure body for outer
    Use (..)
    In Proc outer = (launcher :Task) -> void:
       ( Let inner :Task = create_task (outer, 1)
               \declare task object for inner
               \master is outer, no of entries is 1
         Let inner_e_var :ptr Int = GenPtr SomeInt
               \declare variable for entry parameter, type Int
         Let innerbody =
               \declare procedure body for task object
               Use (inner, inner_e_var, ..)
                   \non locals include Task and variable
               In Proc inner = (launcher :Task) -> void:
                  ( am_activated (inner, launcher);
                    \signal succesful activation to launcher
                    ..
                    ..
                    \body of inner
                    \uses inner and inner_e_var
                    \when accepting entry E
                    ..
                    ..
                    \completed
                    \implicit call of terminate
                  )
               Enduse
         In am_activated (outer, launcher);
            \signal succesful activation to launcher
            launch_tasks (outer, PackVec (inner, innerbody));
            \activation task inner and its body; launcher is outer
            \wait for until inner activated
            ..
            ..
            \body of outer
            \uses inner and inner_e_var when calling entry INNER.E
            ..
            ..
         Ni
         \completed
         \implicit call of terminate
       )
    Enduse
In launch_outer(outer, outerbody) Ni
   \run outer in separate process
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet ....UNCLASSIFIED....................

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memorandum 4074		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
	Royal Signals and Radar Establishment St Andrews Road, Malvern, Worcestershire WR14 3PS
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

ADA TASKING IN TEN15

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref
Tombs D J			1987.10	26

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

An Ada compiler developed at RSRE uses the Ten15 abstract machine as its target. This paper describes in detail the Ten15 implementation of the tasking support for this compiler.

S80/48

ATE

LMED

8