Rendezvous Primitives for Intertask Communication on VAX / VMS

Tengvall, Freddy

1982

Document Version:
Publisher's PDF, also known as Version of record

CODEN:LUTFD2/(TFRT-7234)/1-32/(1982)

RENDEZVOUS PRIMITIVES FOR INTERTASK COMMUNICATION ON VAX/VMS

FREDDY TENGVALL

DEPARTMENT OF AUTOMATIC CONTROL

LUND INSTITUTE OF TECHNOLOGY

MARCH 1982

LUND INSTITUTE OF TECHNOLOGY
DEPARTMENT OF AUTOMATIC CONTROL
Box 725
S 220 07 Lund 7 Sweden

Document name
REPORT

Date of issue
MARCH 1982

Document number
CODEN:LUTFD2/(TFRT-7234)/1-32/(1982)

Author(s)
FREDDY TENGVALL

Supervisor
HILDING ELMQVIST

Sponsoring organization
SWEDISH STATE POWER BOARD

Title and subtitle

RENDEZVOUS PRIMITIVES FOR INTERTASK COMMUNICATION ON VAX/VMS

Abstract

This report describes the design and implementation of primitives that support concurrent programming on VAX/VMS. The primitives provide for dynamic task activation, task scheduling and intertask communication. The intertask communication primitives are designed to correspond to the rendezvous concept in ADA. Entry call and accept statements are the primary means of intertask communication. The primitives has been written in VAX-11 Pascal.

Key words
Concurrent Programming, ADA, Intertask Communication, Rendezvous, Pascal

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title

ISBN

Language
English

Number of pages
32

Recipient's notes

Security classification

Distribution: The report may be ordered from the Department of Automatic Control or borrowed through the University Library 2, Box 1010, S-221 03 Lund, Sweden, Telex: 33248 lubbis lund.

TABLE OF CONTENTS

1. INTRODUCTION 3
1.1 Concurrency in ADA 3

2. DESCRIPTION OF PRIMITIVES 5
2.1 The intertask parameter format 6
2.2 Task activation and scheduling 7
2.3 Communication and synchronization 8
2.4 An example of intertask communication 10

3. IMPLEMENTATION 14
3.1 System services on VAX/VMS 14
3.2 Intertask communication on VAX/VMS 15
3.3 Intertask communication using mailboxes 16
3.4 Rendezvous on VAX/VMS 16
3.5 Comments to the Entry Group concept 18
3.6 Implementation of primitives 19

4. CONCLUSIONS 21

5. ACKNOWLEDGEMENTS 21

6. REFERENCES 22

7. APPENDIX A:1
Package Rendezvous A:1
Package VMS A:5
Package BasicDecl A:10

1. INTRODUCTION

This report describes the design and implementation of primitives for concurrent programming on VAX/VMS. The study is carried through as a project to develop software that supports intertask communication.

The intertask capabilities provided at the user application level, are in most cases limited. On a lower level, however, there are usually different technics available, by using operating system requests.

The intention of this work has been to design and develop a mechanism suitable for the user application level. Ideally, a concurrent programming environment should allow a user to develop concurrent programs without being concerned by any details of the underlying software and hardware.

The approach is to provide a set of primitive operations or intrinsic functions in a high level language, which support concurrent programming, in terms of dynamic task activation, task scheduling and intertask communication on VAX/VMS.

While waiting for ADA compilers to be available, the intertask primitives have been designed and developed according to the rendezvous concept in ADA. The task structure has been designed to correspond to the task concept in ADA.

Concurrent programming in ADA is described below in order to be referred to when discussing the relationship between ADA and the primitives.

1.1 Concurrency in ADA

For real-time applications, ADA provides facilities for multitasking - that is, for logically parallell activities that can cooperate in carefully controlled ways [1,3]. Tasks are the modular units for concurrent programming and entities that may operate in parallell. Communication among concurrently executing tasks is supported by a novel mechanism called the rendezvous mechanism which requires synhronization between a called task and a calling task whenever information is transferred between them.

A task is divided into a specification part and a body. The task specification contains entry declarations and define the procedure-like calls that can be made to communicate with the task. The task body contains the code and variables - that is, the internal state - defining the behavior of the task. The task specification and the task body is illustrated below.

TASK SPECIFICATION
may include type specifications subprogram specifications exception specification entry specifications but may not include variable specifications package specifications

TASK BODY
contains hidden local declarations which are elaborated when the task is initiated and a statement sequence which is executed when the task is initiated and accept statements which must be executed in order to accept entry calls from other tasks

In ADA, entries are the principal means of communication between tasks. Tasks may have entries which may be called by other tasks. Synchronization is achieved by rendezvous between a task issuing an entry call and a task accepting the call.

- o If a calling task issues an entry call before a corresponding accept statement is reached by the task owning the entry, the execution of the calling task is suspended.

- o If a task reaches an accept statement prior to any call of that entry, the execution of the task is suspended until such a call occurs.

If several tasks call the same entry before a corresponding accept statement is reached, the calls are queued. There is one queue associated with each entry. The calls are processed in order of arrival.

The principal structure of a task accepting entry calls from a calling task is illustrated below.

```
task ILLUTASK is          -- this is the task specification part
   ...
  entry ENTRYA (...)
  entry ENTRYB (...)
  entry ENTRYC (...)
   ...

task body ILLUTASK is     -- this is the task body part
   ...
begin
 ...
accept ENTRYA (...) do
 ...
end;
 ...
select
  when CONDITIONB =>
    accept ENTRYB (...) do
     ...
    end;
or
  when CONDITIONC =>
    accept ENTRYC (...) do
     ...
    end;
or
  ...                     -- other accept alternatives
end select;
 ...
end ILLUTASK;
```

In this example, the task specification declares the entries ENTRYA, ENTRYB and ENTRYC for use by other tasks. The code for the task consists of an accept statement for entry 'ENTRYA', a select statement with two "when clauses" containing accept statements for entry 'ENTRYB' and 'ENTRYC'.

2. DESCRIPTION OF PRIMITIVES

This section describes a set of procedures called the primitives, performing dynamic task creation, task scheduling and intertask communication. They have been written in VAX-11 Pascal and implemented on VAX/VMS.

The procedure 'ActivateTask' allows a task to activate another task, called a subtask. Subtasks are related to their creators in a tree-like structure. A task is terminated when either all it's statements are executed or when the procedure 'Terminate' is called (or when an exception occurs). The procedure 'Delay' makes the calling task wait a specified time interval.

By a call to the procedure 'ConnectTask', the intertask connection set up is done. Entry declarations are performed by the procedure 'Entry'. The procedure 'CallEntry' issues an entry call to a specified task and the 'Accept' procedure enables a task to accept an entry call. Choice among several entry calls based on a dynamic guard is performed by the procedure 'SelectAccept'. 'EndAccept' terminates the accept statement which causes rendezvous completion.

2.1 The intertask parameter format

The intertask parameter format refer to the concept of how to define an entry and its corresponding formal parameter part. In ADA an entry and its formal parameter part is given in the entry declaration statement. In this implementation an entry is defined as an element of an enumerated type 'entries' (see the example below) and declared by a call to the procedure 'Entry' (see the procedure description).

Each entry belongs to a unique identity, called an entry group. The entry group defines a concept that consists of a number of entries related to a task that ownes entries. That is - each task owning entries are assigned an identity referred to by the calling tasks. A specific entry call is qualified by both the entry and the entry group.

Entry groups are declared as elements of an enumerated type 'GroupsOfEntries' and it is recommended to declare them together with the parameter format (ref. to the example at the end of this chapter). The entry group concept is also discussed in the implementation section.

The intention has been to design one set of primitives that could handle all relevant parameter formats. In order to facilitate the management of these parameter formats between cooperating tasks, one basic common parameter format structure has been utilized. A record structure with variants enables different parameter types within one frame. This means that all tasks performing intertask communication according to the rendezvous concept have to refer to the same parameter record.

The fixed part consists of the parameters 'rejected' and 'invalid' of type 'boolean' which are only used internally. The tag-field 'entry' of type 'entries' is set by the user when making an entry call. It is used by 'Accept' and 'SelectAccept' to indicate the current entry variant of the record. Due to the implementation the record size (in bytes) must be supplied by the user explicitly.

Accordingly, the parameter structure can easily be expanded by insertions of new entry variants. The type declarations may then be expressed in the following way:

```
entries = (entryA, entryB, ... , entryN);
GroupsOfEntries =(EntryGroupU, EntryGroupV, ... , EntryGroupZ);

parameters = record
  rejected, invalid: boolean;
  case entry: entries of
    entryA:  (...);
    entryB:  (...);
     ...
    entryN:  (...);
end;
```

2.2 Task activation and scheduling

A subtask is activated by calling the procedure 'ActivateTask'.

```
procedure ActivateTask (task: TaskImage; name: string10);
```

task — represents the filename and extensions of the VAX/VMS image to be activated. The default extension for images is .EXE. TaskImage is a string of ten characters.

name — user defined name of the task. Name is a string of ten characters.

'ActivateTask' creates the subtask 'task', related to its creator in a tree-like structure. It receives a portion of the creating tasks resource quotas and must terminate before the creating task.

When a subtask is created, the value of any deductible quota is subtracted from the total value the creator has available, and when the subtask is deleted, the unused portion of any deductible quota is added back to the total available to the creator.

The input, output and error files for an activated subtask are assigned the logical names INPUT, OUTPUT and ERROR respectively and the relevant physical unit may be assigned by the user at DCL command level.

A task is terminated when all it's statements are executed or when the procedure 'Terminate' is called.

```
procedure Terminate;
```

Normal termination of a task in ADA occurs when it reaches the end of it's task body and when all locally declared tasks (if any) have terminated their execution.

The procedure 'Delay' makes the calling task wait a specified time interval, i.e. the calling task is put in a hibernate state until the time has elapsed.

```
procedure Delay (time: TimeString);
```

TimeString	an ascii character string descriptor specifying the time interval with the following (string13) format 'd hh:mm:ss.cc'. Example: '0 00:05.20.00' (wait 5 min and 20 sec)

2.3 Communication and synchronization

Before a rendezvous between two tasks occurs, each task must be prepared. That is - the task issuing entry calls must call the procedure 'ConnectTask' and the accepting task must call 'Entry' (see below) before any usage of the rendezvous primitives.

```
procedure ConnectTask (EntryGroup: GroupsOfEntries);
```

EntryGroup	name of an entry group to which the entry belong

By calling 'ConnectTask' a connection to an entry group 'EntryGroup' is made, i.e. a reference to the actual entry group is established. The entry group can now subsequently be called from this task. This procedure has no corresponding statement in ADA.

The entry declaration is elaborated by the procedure 'Entry'.

```
procedure Entry (EntryGroup: GroupsOfEntries;
                 entries: EntrySet);
```

EntryGroup	see previous decl.
entries	set of entries in the entry group

The procedure 'Entry' defines the entries 'entries' in the entry group 'EntryGroup'. In order to avoid that invalid entries are used in conjunction with an entry group, the user must explicitly specify all entries that will be referenced by accept statements. Otherwise, an exception is raised at run time, causing the task to be aborted. This procedure together with the parameter record specification corresponds to the entry declaration statement in ADA.

An entry call is issued by calling 'CallEntry'.

```
procedure CallEntry (EntryGroup: GroupsOfEntries;
                     entry: entries;
                     var par: parameters);
```

EntryGroup	see previous decl.
entry	name of the entry
par	parameters that are transferred between the task issuing the call and the task owning the entry

The procedure 'CallEntry' issues an entry call to the entry 'entry' in the entry group 'EntryGroup'. The called task does not know the name of the caller. Entry calls for a given entry name may occur faster than they can be handled by accept statements.

In this case they are stored in a queue associated with the entry name and handled in order of arrival by accept statements of the called tasks. If a task issues an entry call with an invalid entry - that is, the entry is not declared in the task that accept the entry, an exception occurs, causing the calling task to be aborted.

The 'Accept' procedure enables a task to wait for a specific entry call.

```
procedure Accept (EntryGroup: GroupOfEntries;
                  entry: entries;
                  var par: parameters);
```

In many cases we cannot predict the order in which entry calls will occur and wish to allow a task to base its next action on which entry calls (if any) are waiting. Choice among several entry calls is accomplished by the procedure 'SelectAccept'.

```
procedure SelectAccept (EntryGroup: GroupsOfEntries;
                        accept: EntrySet);
                        var par: parameters;
```

accept set of valid entries which may be modified (dynamically) during program execution

When 'EndAccept' is executed the accept statement is terminated which causes rendezvous completion.

```
procedure EndAccept (EntryGroup: GroupsOfEntries;
                     par: parameters);
```

The sequence of statements between 'Accept' ('SelectAccept') and 'EndAccept' is a critical region because the calling task cannot execute while these statements are executed.

The accept statement in ADA has the following syntax

```
accept entry-name [formal part] do
   sequence-of-statements
end [identifier]
```

The 'Accept' and 'EndAccept' procedures corresponds to accept and end in the previous accept statement in ADA. The procedures 'SelectAccept' and 'EndAccept' corresponds in ADA to a select statement with "when clauses" containing accept statements.

2.4 An example of intertask communication

To make it easier to use the primitives, an example is given below, including three concurrent tasks that illustrates intertask communication. The example defines a buffering task to smooth variations between the speed of output of a producing task and the speed of input of some consuming task [2]. The task graph is showed in figure 5.

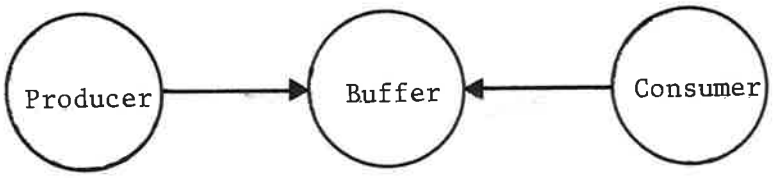

Figure 1. Task graph showing the producing, consuming and buffering task.

In ADA logically related units may be allocated into modules, called packages. In order to follow the package concept a preprocessor has been used. The preprocessor sorts and merges Pascal files prepared with special keywords into one ordinary Pascal file. In the Appendix the package concept is utilized.

In our example the intertask parameters are declared in the package 'ParameterFormat' (see below). It is recommended to declare the entry groups 'GroupsOfEntries' in the same package since the declarations are "global" to all tasks.

```
package ParameterFormat is

  -- User supplied package containing parameter format
  -- declarations used by the consumer, producer and
  -- buffer task.

.CONST
       ParameterSize = 4; {record size in bytes}

.TYPE
      GroupsOfEntries = (BUFFER);
      entries = (read, write);

      parameters = record
        rejected, invalid: boolean;
        case entry: entries of
          read, write: (ch: char);
      end;
.END
```

By the preprocessor some of the basic principals, of the task concept in ADA, may be followed e.g. separation of specification and body. This is illustrated in the buffering task. The producing and consuming task are illustrated without.

For instance, the producing task may contain the statements

```
begin {producer}
ConnectTask (BUFFER);
while true do
  begin {loop}
  {-- produce the next character 'charac'}
  param.ch := charac;
  CallEntry (BUFFER, write, param);
  end {loop}
end; {producer}
```

and the consuming task may contain the statements

```
begin {consumer}
ConnectTask (BUFFER);
while true do
  begin {loop}
  CallEntry (BUFFER, read, param);
  charac := param.ch
  {-- consume the character 'charac'}
  end {loop}
end; {consumer}
```

The buffering task contains an internal pool of characters processed in a round-robin fashion. The pool has two indices, InIndex denoting the space for the next character and an OutIndex denoting the space for the next output character. See figure 6.

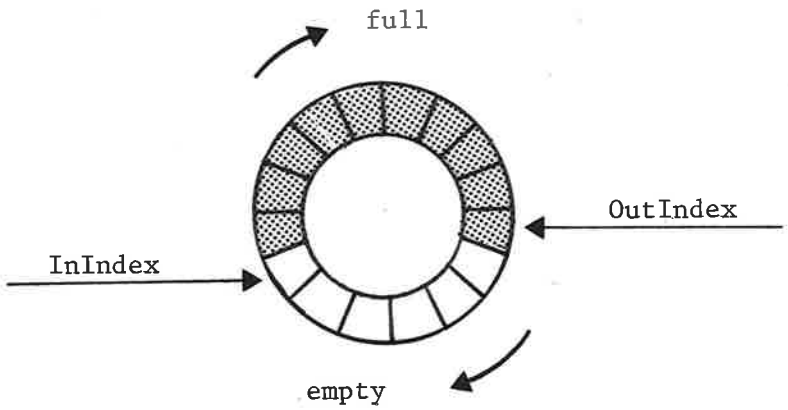

Figure 6. A cyclic character buffer pool with two indices, InIndex and OutIndex.

The buffering task with the keywords inserted may contain the following statements:

```
task BUFFER is

  uses Rendezvous, ParameterFormat

.INIT
  Entry (BUFFER, [read, write]);

.PROGRAM
  program BUFFER (output);
.END
```

```
task body BUFFER is

.CONST
       PoolSize = 100;

.VAR
     pool: array [1..PoolSize] of char;
     count: 0..PoolSize;
     InIndex, OutIndex: 1..PoolSize;
     AcceptSet: EntrySet;
     param: parameters;

.INIT
     count := 0;
     InIndex := 1; OutIndex := 1;

.MAIN
  begin {buffer}
   while true do
    begin {loop}
    AcceptSet := [read, write];
    if count = 0 then
      AcceptSet := AcceptSet - [read]
    if count = PoolSize then
      AcceptSet := AcceptSet - [write]
    SelectAccept (BUFFER, AcceptSet, param);
    case param.entry of
      write:   begin
               pool [InIndex] := param.ch;
               EndAccept (BUFFER, param);
               InIndex := InIndex mod PoolSize + 1;
               count := count + 1;
               end;
      read:    begin
               param.ch := pool [OutIndex];
               EndAccept (BUFFER, param);
               OutIndex := OutIndex mod PoolSize + 1;
               count := count - 1;
               end
      end {case}
    end {loop}
  end; {buffer}
.END
```

3. IMPLEMENTATION

The implementation is based upon the operating system VMS (Virtual Memory System). The primitives are written in VAX-11 Pascal. Some of them act as an interface between existing operating system requests and user programs. Others consist of a combination of requests.

The rendezvous primitives are allocated into package Rendezvous. Package VMS provides routines that supports the Rendezvous Package and acts as an interface to VAX/VMS system services. Package BasicDecl contains some simple declarations to support the previous packages. The code is listed in appendix A.

VAX-11 Pascal provides predefined procedures to perform input and output operations to mailboxes by file variables, but these procedures implies quite a lot of restrictions in conjunction with the implementation. For example: file variables must be declared as externals. Instead the queued input and output (Q I/O) read and write requests has been used to perform the equivalent I/O functions.

The advantages that we achieve by this solution are: an increasing flexibility when implementing new primitives and a reduced I/O overhead i.e. better performance at run time. The disadvantage is that the message size must be known and supplied by the user, but this causes problably no problems.

The details that are needed for understanding the implementation are described below for readers that are not familiar with the VMS operating system services and the calling conventions from VAX-11 Pascal programs.

3.1 System services on VAX/VMS

System services are procedures used by the operating system to control resources available to tasks, to provide for communication among tasks, and to perform basic operating system functions, such as the coordination of input/output operations [4].

Although most system services are used primarily by the operating system itself on behalf of logged-on users, many are available for general use and provide techniques that can be used in application programs.

When a system service that uses a resource controlled by a quota is called, the task quota for that resource is checked. When the task issues a call to a system service that is protected by privilege , the privilege list is checked.

Any VAX/VMS system service can be declared as an external function or procedure and then be called from a Pascal program. When calling non-pascal procedures, for example system services, special calling conventions must be applied [6,7].

Non-Pascal procedures require arguments as addresses, immediate values or descriptors. The VAX-11 procedure calling standard defines three mechanisms by which arguments are passed to procedures.

- o By-reference -- the argument list entry is the address of the value
- o By-immediate-value -- the argument list entry is the value
- o By-descriptor -- the argument list entry is the address of a descriptor of the value

3.2 Intertask communication on VAX/VMS

The VAX/VMS operating system provides intertask communication facilities for synchronizing execution, for sending messages and for sharing common data [5,8]. The three communication techniques utilized by cooperating tasks are:

- o common event flags
- o mailboxes
- o shared areas of memory

There are four major classes of primitives for concurrent programming:

- o semaphores
- o monitor procedures
- o messages
- o rendezvous

The last enumerated class is the one that has been used, as discussed earlier, to implement the intertask communication. Rendezvous according to the rules of ADA has been implemented by using mailboxes combined with common event flags.

Event flags are posting status bits used by certain I/O system services to indicate the completion or occurence of an event.

A mailbox is a buffer in virtual memory that is treated exactly as if it were a record oriented I/O device. A task creates a mailbox into which other tasks can write data, and from which it can read data. A mailbox has full device context. A mailbox is created for a task when the task issues a Create Mailbox request. Once a mailbox has been

created, tasks can open the mailbox for reading and writing. Messages are sent to a mailbox by programs issuing a Q I/O write command.

3.3 Intertask communication using mailboxes

Mailboxes provide a controlled and synchronized method for tasks to exchange data [8]. Write QIO requests are used to transfer data from a task to a mailbox. To obtain messages in a mailbox, written by other tasks the read mailbox QIO requests are used. See figure 1.

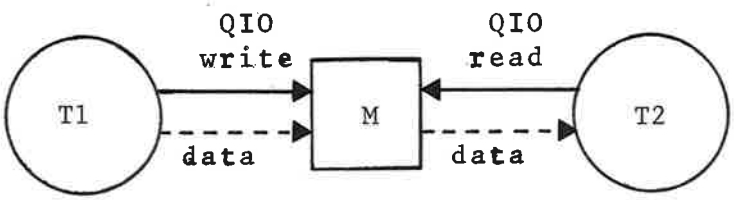

Figure 2. QIO requests and data transfer between two cooperating tasks, T1 and T2.

Suppose that task T1 writes a message to be read by task T2. Two possibilities exist.

1. Task T1 issues a write request before task T2 issues a read request. The write QIO request is queued during the wait for task T2 to issue a read QIO request (task T1 is placed in a waiting state until the QIO read request issued by task T2 is completed). When this request occurs, the data is transferred form task T1 to task T2 to complete the I/O operation.

2. If task T2 issues a read request before task T1 issues a write request, task T1 finds a waiting request in the mailbox. The data is transferred and the I/O operation is completed immediately.

3.4 Rendezvous on VAX/VMS

A rendezvous is referred to as the process of synchronization between an entry call in a calling task and an accept statement in a called task. The rendezvous concept has been implemented on VAX/VMS by using the mailbox facility combined with common event flags, and primitives for sending and receiving messages.

To facilitate the discussion and implementation of rendezvous through mailboxes, the entry_group concept is introduced. An entry group is referred to as an environment consisting of two mailboxes, and corresponds to the context, in which tasks cooperates. This concept is illustrated in figure 2.

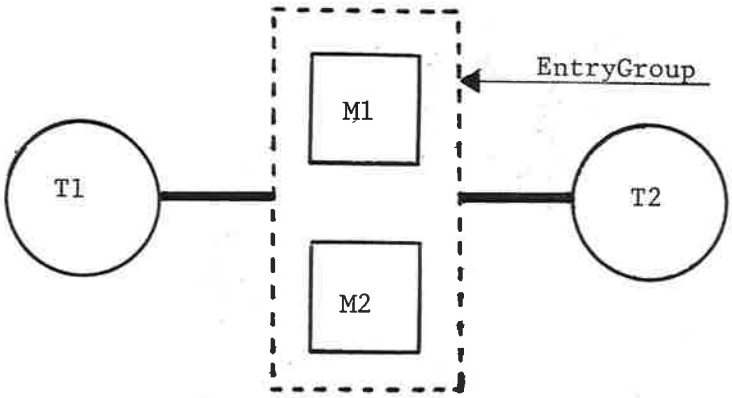

Figure 3. Illustration of the entry group concept.

Consequently the implementation of the rendezvous, in terms of QIO requests and datatransfer, is illustrated by figure 3.

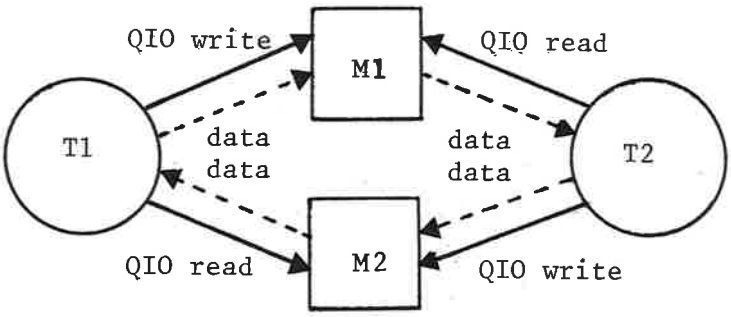

Figure 4. Implementation of rendezvous on VAX/VMS using mailboxes.

An entry call issued by task T1 to task T2 would be equivalent to a write QIO request to mailbox M1 and read QIO request to mailbox M2. Analoguous the accept statement in task T2 is initialized by issuing a read QIO request to mailbox M1 and terminated when a QIO write request to mailbox M2 has been received by task T1. Virtually, this structure corresponds to an intertask communication viewed by figure 4.

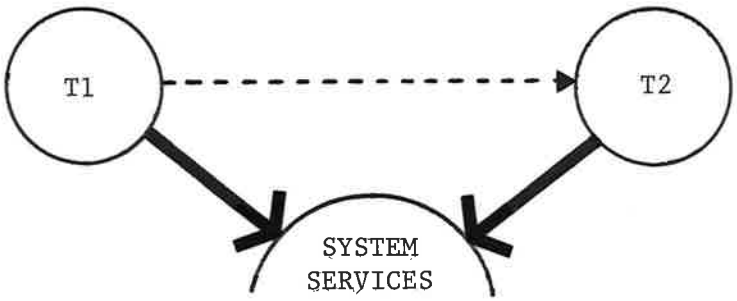

Figure 5 Virtual view of the rendezvous concept.

Referring to the example described in the previous section the task graph may be modified to illustrate the intertask communication in terms of mailboxes and QIO requests. Accordingly, the mailboxes correspond to the entry group 'BUFFER'.

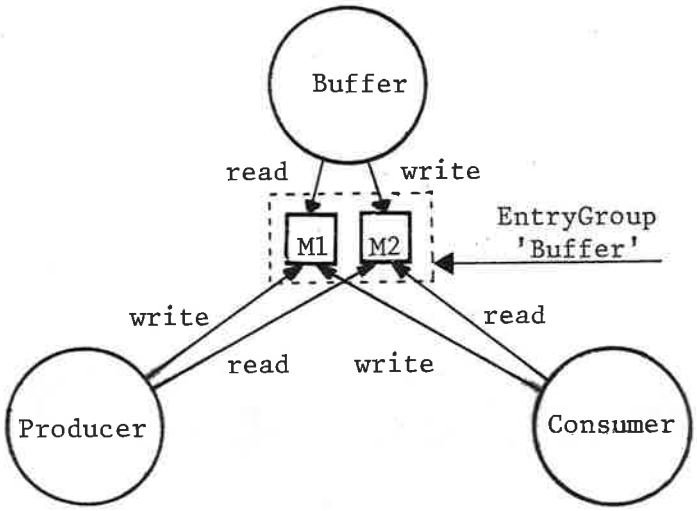

Figure 6 The producing, consuming and buffering task illustrated in terms of mailboxes and QIO requests.

3.5 Comments to the Entry Group concept

Entries are the principal means of communication and each entry belongs to an entry group. An entry group may have an arbitrary number of entries. A task may have several entry groups, but each entry group that is introduced affects the resource quota, independently of the numbers of entries that belongs to it. At present the maximum numbers of entry groups are restricted to 10, but if necessary, it could be expanded.

On the other hand, an increasing number of entries, in an entry group, reduces the efficency and performance of the accept statement, on account of an increasing overhead. This is discussed in the next section.

The choice of entry groups and the corresponding entries is not distinct but consequently a question of compromise between resources and performance.

3.6 Implementation of primitives

Before any entry calls or accept statements may be executed the affected entry group(s) must be initialized. The initialization implies the creation of two mailboxes, denoted by SendBoxes[EntryGroup] and ReceiveBoxes[EntryGroup] respectively, and also that a so called valid entry set is defined. Accordingly, each task must execute the create mailbox system request ($CREMBX) and the task owning the entries define a relevant entry mask.

The entry mask declares which entries that belongs to the accepting task and are referred to by accept statements. To handle illegal entry calls a boolean variable 'invalid' has been included in the intertask parameter structure. In such cases 'invalid' is set by the accept procedures and an exception is raised by the procedure 'CallEntry'. That is, when a task issues an entry call to a task whose corresponding entry is not defined within the entry mask, the calling task will be notified by 'invalid' causing the calling task to be aborted.

If several QIO write requests are issued to a mailbox before the corresponding QIO read requests are performed, the requests are queued and processed in order of arrival.

In order to access a desired entry point in a called task, each intertask parameter structure is assigned a unique identity, the entry name. Aided by this identity, an appropriate action is performed, by the task owning the entry.

To indicate whether an entry call proceeds successfully or not a boolean variable 'rejected' has been included in the intertask parameter structure. If an entry call occurs that do not satisfy the valid accept conditions, the entry call is rejected, indicated by the mentioned variable.

When an entry call is rejected, then the issuing task is placed in a wait state, waiting on an event flag, associated with the entry group, to be set by the task owning the entry. When the flag is set, the entry call is repeated. This cycle continues until the call is accepted.

The event flag acts as a collector, collecting rejected entry calls. By introducing event flags, rejected entry calls are prevented from being repeated before the relevant entry call has been accepted.

This implies a better performance on account of an

increasing throughput of queued entry calls and less overhead.

The optimal solution, however, based on this mailbox concept, would probably be one where each entry is assigned an unique entry group. But this technique causes to much overhead on account of system resources and is not recommended.

4. CONCLUSIONS

Rendezvous according to the rules of ADA has been implemented by using primitives for sending and receiving messages through mailboxes complemented by common event flags. To facilitate the management of different parameter types between cooperating tasks, one basic common message format structure has been utilized. The intention has been to illustrate the principals of one possible method of implementation on VAX/VMS. The design philosophy has shown that additional primitives e.g. timed and entry (accept) calls can be implemented on VAX/VMS without causing to much effort.

5. ACKNOWLEDGEMENTS

I would like to thank my adviser Hilding Elmqvist for initiating this work and for his support. I also want to thank Leif Andersson, Tommy Essebo and Tomas Schonthal for their very helpful contributions. Finally, i would like to express my gratitude to the Swedish State Power Board for sponsoring this work.

6. REFERENCES

[1] 'Reference Manual for the ADA Programming Language', United States Department of Defence, July 1980.

[2] Brinch Hansen, P., 'Operating System Principles', Prentice Hall, 1973.

[3] Wegner, P., 'Programming with ADA', Prentice Hall, 1980.

[4] 'VAX/VMS System Services Reference Manual', Digital Equipment Corporation, 1980.

[5] 'VAX/VMS Real-Time User's Guide', Digital Equipment Corporation, 1980.

[6] 'VAX-Pascal Language Reference Manual', Digital Equipment Corporation, 1980.

[7] 'VAX-11 Pascal User's Guide', Digital Equipment Corporation, 1980.

[8] 'VAX/VMS I/O User's Guide', Digital Equipment Corporation, 1980.

Appendix
package Rendezvous

```
package Rendezvous is

  ConnectError, InvalidEntry,
  InvalidCall, InvalidAcceptSet, GroupsExceeded: ExceptionSetRV;

  uses VMS, ParameterFormat, BasicDecl

  -- This package provides primitives for intertask communication
  -- according to the rendezvous concept in ADA.

  -- Author: Freddy Tengvall
  -- Date:   1981-10-11


.FORWARD
procedure ConnectTask (EntryGroup: GroupsOfEntries);
procedure Entry (EntryGroup: GroupsOfEntries;
                 entries: EntrySet); forward;
procedure CallEntry (EntryGroup: GroupsOfEntries;
                     entry: entries;
                     var par: parameters); forward;
procedure Accept (EntryGroup: GroupsOfEntries;
                  entry: entries;
                  var par: parameters); forward;
procedure SelectAccept (EntryGroup: GroupsOfEntries;
                        accept: EntrySet); forward;
                        var par: parameters;
procedure EndAccept (EntryGroup: GroupsOfEntries;
                     var par: parameters); forward;
.END

-------------------------------------------------------------

package body Rendezvous is

.CONST
       MaxEntryGroups = 10;
       ref = 63;

.TYPE
      EntrySet = set of entries;
      ExceptionSetRV = (ConnectError, InvalidEntry, InvalidCall,
                        InvalidAcceptSet, GroupsExceeded);

.VAR
     SendName, Receivename: string5;
     SendBoxes, ReceiveBoxes: array [GroupsOfEntries] of chantype;
     ConnectSet: set of GroupsOfEntries;
     EntryMask: array [GroupsOfEntries] of EntrySet;


.INIT
     ConnectSet := [];
```

Appendix
package Rendezvous

```
.PROCEDURE

{----------}

procedure RaiseRendezvous (exception: ExceptionSetRV);

{
-- Exceptionhandler for package Rendezvous
}

  begin {RaiseRendezvous}
  case exception of
    ConnectError:       writeln ('Reference Error! Task not connected!');
    InvalidEntry:       writeln ('Invalid Entry! Entry not defined!');
    InvalidCall:        writeln ('Entry not declared in accepting task!');
    InvalidAcceptSet:   writeln ('Invalid AcceptSet! Entry not defined!');
    GroupsExceeded:     writeln ('Entry Group Error! Max   exceeded!');
    end; {case}
  halt
  end; {RaiseRendezvous}

{----------}

procedure ConnectTask {EntryGroup: GroupsOfEntries};

  begin {ConnectTask}
  if ord (EntryGroup) >= MaxEntryGroups then
    RaiseRendezvous (GroupsExceeded);
  ConnectFlags ('FLAGS', ref + 1);
  SendName := 'SBOX ';
  ReceiveName := 'RBOX ';
  SendName [5] := chr (ord ('0') + ord (EntryGroup));
  ReceiveName [5] := chr (ord ('0') + ord (EntryGroup));
  CreateMailbox (SendName, SendBoxes [EntryGroup] );
  CreateMailbox (ReceiveName, ReceiveBoxes [EntryGroup]);
  ConnectSet := ConnectSet + [EntryGroup];
  end; {ConnectTask}

{----------}

procedure Entry {EntryGroup: GroupsOfEntries;
                 entries: EntrySet};

  begin {entry}
  ConnectTask (EntryGroup);
  EntryMask [EntryGroup] := entries;
  end; {entry}

{----------}

procedure CallEntry {EntryGroup: GroupsOfEntries;
                     entry: entries;
                     var par: parameters};

  begin {CallEntry}
```

Appendix
package Rendezvous

```
   if not (EntryGroup in ConnectSet) then
     RaiseRendezvous (ConnectError);
   par.invalid := false;
   par.entry := entry;
   PutBox (SendBoxes [EntryGroup], par);
   GetBox (ReceiveBoxes [EntryGroup], par);
   if par.invalid then
     RaiseRendezvous (InvalidCall)
   else
     while par.rejected do
       begin
       WaitEv (ref + ord (EntryGroup));        {Wait for next Accept}
       PutBox (SendBoxes [EntryGroup], par);  {or SelectAccept.}
       GetBox (ReceiveBoxes [EntryGroup], par);
       end; {while}
    end; {CallEntry}

{----------}

procedure Accept {EntryGroup: GroupsOfEntries;
                  entry: entries;
                  var par: parameters};

  begin {Accept}
  if not (EntryGroup in ConnectSet) then
    RaiseRendezvous (ConnectError);
  if not (entry in EntryMask [EntryGroup]) then
    RaiseRendezvous (InvalidEntry);
  SetEv (ref + ord (EntryGroup));    {Trig rejected calls}
  ClrEv (ref + ord (EntryGroup));    {to be tried again.}
  GetBox (SendBoxes [EntryGroup], par);
  if not (par.entry in EntryMask [EntryGroup]) then
    par.invalid := true
  else
    while not (par.entry = entry) do
      begin
      par.rejected := true;
      PutBox (ReceiveBoxes [EntryGroup], par);
      GetBox (SendBoxes [EntryGroup], par);
      end; {while}
  end; {Accept}

{----------}

procedure SelectAccept {EntryGroup: GroupsOfEntries;
                        accept: EntrySet};
                        var par: parameters;

  begin {SelectAccept}
  if not (EntryGroup in ConnectSet) then
    RaiseRendezvous (ConnectError);
  if ((EntryMask [EntryGroup] + accept) <> EntryMask [EntryGroup]) then
    RaiseRendezvous (InvalidAcceptSet);
  SetEv (ref + ord (EntryGroup));    {Trig rejected calls}
  ClrEv (ref + ord (EntryGroup));    {to be tried again}
```

Appendix
package Rendezvous

```
  GetBox (SendBoxes [EntryGroup], par);
  if not (par.entry in EntryMask [EntryGroup]) then
    par.invalid := true
  else
    while not (par.entry in accept) do
      begin
      par.rejected := true;
      PutBox (ReceiveBoxes [EntryGroup], par);
      GetBox (SendBoxes [EntryGroup], par);
      end; {while}
  end; {SelectAccept}

{----------}

procedure EndAccept {EntryGroup: GroupsOfEntries;
                     var par: parameters};

  begin {EndAccept}
  if not (EntryGroup in ConnectSet) then
    RaiseRendezvous (ConnectError);
  par.rejected := false;
  PutBox (ReceiveBoxes [EntryGroup], par);
  end; {EndAccept}

{----------}

.END
```

Appendix
package VMS

```
package VMS is

  ActivateError, TerminateError, CreMailError,
  WaitEvError, SetEvError, ClearEvError, DelayError,
  HiberError, EvClustError, GetError, PutError: ExceptionsSetVMS

  uses BasicDecl

  -- This package provides routines that supports
  -- the Rendezvous package and acts as an interface
  -- to VAX/VMS system services.

  -- Author: Freddy Tengvall
  -- Date:   1981-10-11


.TYPE
      MaskType = packed array [0..63] of boolean;

.FORWARD
procedure ActivateTask (task: TaskImage; name: string10); forward;
procedure Terminate; forward;
procedure CreateMailbox (mailbox: string5;
                         var channel: ChanType); forward;
procedure WaitEv (EvFlag: event); forward;
procedure SetEv (EvFlag: event); forward;
procedure ClrEv (EvFlag: event); forward;
procedure Delay (time: string13); forward;
procedure ConnectFlags (cluster: string5; efn: event); forward;
procedure GetBox (BoxChan: ChanType; var par: parameters); forward;
procedure PutBox (BoxChan: ChanType; par: parameters); forward;

.PROCEDURE
function sys$creprc (%immed pidadr: integer;
                     %stdescr image: string10;
                     %stdescr input: string5;
                     %stdescr output: string6;
                     %stdescr error: string5;
                     var prvadr: masktype;
                     %immed quota: integer;
                     %stdescr prcnam: string10;
                     %immed baspri: integer;
                     %immed uic: integer;
                     %immed mbxunt: integer;
                     %immed stsflg: integer):
                     integer; extern;
function sys$exit (var code: integer): integer; extern;
function sys$crembx (%immed prmflg: integer;
                     var chan: ChanType;
                     %immed maxmsg, bufquo, promsk, acmode: integer;
                     %stdescr logname: string5):
                     integer; extern;
function sys$waitfr (%immed flag: event): integer; extern;
function sys$setef (%immed flag: event): integer; extern;
function sys$clref (%immed flag: event): integer; extern;
```

Appendix
package VMS

```
 function sys$bintim (%stdescr asciitime: string13;
                      var binarytime: quad):
                      integer; extern;
 function sys$schdwk (%immed pidadr: integer;
                      %immed prcnam: integer;
                      daytim, reptim: quad):
                      integer; extern;
 function sys$hiber: integer; extern;
 function sys$ascefc (%immed efn: event;
                      %stdescr name: string5;
                      %immed prot, perm: integer):
                      integer; extern;
 function sys$qiow (%immed efn: integer;
                    %immed chan: ChanType;
                    %immed func: integer;
                    var iosb: quad;
                    %immed astadr: integer;
                    astprm: integer;
                    var p1: ;
                    %immed p2, p3, p4, p5, p6: integer):
                    integer; extern;
 .END

 ------------------------------------------------------------

 package body VMS is

 .TYPE
       event = 0..127;
       quad = array [1..2] of integer;
       ChanType = UnsignedInteger;
       TaskImage = string10;
       ExceptionSetVMS = (ActivateError, TerminateError, CreMailError,
                          WaitEvError, SetEvError, ClearEvError,
                          DelayError, HiberError, EvClustError,
                          GetError, PutError);

 .VAR
      status: integer;
      iostat: quad;

 .PROCEDURE

 {----------}

 procedure RaiseVMS (exception: ExceptionSetVMS);

 {
 -- Exceptionhandler for package VMS
 }

  begin {RaiseVMS}
  case exception of
    ActivateError:      writeln ('Activate Error!');
    TerminateError:     writeln ('Termination Error!');
```

```
package VMS


    CreMailError:       writeln ('Create Mailbox Error!');
    WaitEvError:        writeln ('Wait Event Flag Error!');
    SetEvError:         writeln ('Set Event Flag Error!');
    ClearEvError:       writeln ('Clear Event Flag Error!');
    DelayError:         writeln ('Delay Error!');
    HiberError:         writeln ('Hibernate Error!');
    EvClustError:       writeln ('Event Flag Cluster Error!');
    GetError:           writeln ('GetBox Error!');
    PutError:           writeln ('PutBox Error!');
    end; {case}
  halt
  end; {RaiseVMS}

{----------}

procedure ActivateTask {task: TaskImage; name: string10};

var PrivMask: MaskType;
    i: integer;

  begin {ActivateTask}
  for i:= 0 to 63 do
    PrivMask [i] := false;
  PrivMask [15] := true;
  PrivMask [3] := true;
  PrivMask [19] := true;
  status:= sys$creprc (0, task, 'INPUT', 'OUTPUT', 'ERROR',
                       PrivMask, 0, name, 4, 0, 0, 0);
  if not odd (status) then
    RaiseVMS (ActivateError);
  end; {ActivateTask}

{----------}

procedure Terminate;

var code: integer;

  begin {Terminate}
  status := sys$exit (code);
  if not odd (status) then
    RaiseVMS (TerminateError);
  end; {Terminate}

{----------}

procedure CreateMailbox {mailbox:string5;
                         var channel: ChanType};

  begin {CreateMailbox}
  status:= sys$crembx (0, channel, 300, 300, 0, 0, mailbox);
  if not odd (status) then
    RaiseVMS (CreMailError);
  end; {CreateMailbox}
```

Appendix
package VMS

```
{----------}

procedure WaitEv {EvFlag: event};

  {Tests a specific event flag and returns immediately
   if the flag i set, otherwise the process is placed
   in a wait state until the event flag i set.}

  begin {WaitEv}
  status:= sys$waitfr (EvFlag);
  if not odd (status) then
    RaiseVMS (WaitEvError);
  end; {WaitEv}

{----------}

procedure SetEv {EvFlag: event};

  {Sets an event flag in a local or common event flag cluster
   to 1. Any processes waiting for the event flag are made
   runnable.}

  begin {SetEv}
  status:= sys$setef (EvFlag);
  if not odd (status) then
    RaiseVMS (SetEvError);
  end; {SetEv}

{----------}

procedure ClrEv {EvFlag: event};

  {Sets an event flag in a local or common event flag cluster to 0.}

  begin {ClrEv}
  status:= sys$clref (EvFlag);
  if not odd (status) then
    RaiseVMS (ClearEvError);
  end; {ClrEv}

{----------}

procedure Delay {time: string13};

var bintime, zero: quad;

  begin {Delay}
  status := sys$bintim (time, bintime);
  if not odd (status) then
    RaiseVMS (DelayError);
  zero[1] := 0; zero[2] := 0;
  status := sys$schdwk (0, 0, bintime, zero);
  if not odd (status) then
    RaiseVMS (DelayError);
  status:= sys$hiber;
```

```
    if not odd (status) then
      RaiseVMS (HiberError);
    end; {Delay}

  {----------}

  procedure ConnectFlags {cluster: string5; efn: event};

    {Causes a named common event flag cluster to be associated with
     a process. If the named cluster does not exist but the process
     has suitable privilege, the service creates the cluster.}

    begin {ConnectFlags}
    status := sys$ascefc (efn, cluster, 0, 0);
    if not odd (status) then
      RaiseVMS (EvClustError);
    end; {ConnectFlags}

  {----------}

  procedure GetBox {BoxChan: ChanType; var par: parameters};

  const readvblk = 49;

    begin {GetBox}
    status := sys$qiow (0, BoxChan, readvblk, iostat, 0, 0,
                        par, ParameterSize, 0, 0, 0, 0);
    if not odd (status) then
      RaiseVMS (GetError);
    end; {GetBox}

  {----------}

  procedure PutBox {BoxChan: ChanType; par: parameters};

  const writevblk = 48;

    begin {PutBox}
    status := sys$qiow (0, BoxChan, writevblk, iostat, 0, 0,
                        par, ParameterSize, 0, 0, 0, 0);
    if not odd (status) then
      RaiseVMS (PutError);
    end; {PutBox}

  {----------}
  .END
```

Appendix
package BasicDecl

```
package BasicDecl is

  -- This package contains basic declarations

  -- Author: Freddy Tengvall
  -- Date:   1981-10-11

.TYPE
      string2 = packed array [1..2] of char;
      string5 = packed array [1..5] of char;
      string6 = packed array [1..6] of char;
      string9 = packed array [1..9] of char;
      string10 = packed array [1..10] of char;
      string13 = packed array [1..13] of char;

      unsignedinteger = 0..65535;
.END
```