Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

J. Kok

Two Ada mathematical functions packages for use in real time

Department of Numerical Mathematics Report NM-R8512 June

Two Ada Mathematical Functions packages for use in real time

J. Kok
Centre for Mathematics and Computer Science, Amsterdam

Two portable Ada packages are proposed for the provision of basic mathematical functions in a form suitable for real-time processing. These packages satisfy the requirements given in the "Guidelines for the design of large modular scientific libraries in Ada" with regard to services requested by real-time processes.

1980 MATHEMATICS SUBJECT CLASSIFICATION: 69D49, 65-04.
KEY WORDS & PHRASES: Ada, high level language, basic mathematical functions, scientific libraries, portability, real-time processing.
NOTE: This report will be submitted for publication elsewhere.

*) Ada is a registered trademark of the US Government (AJPO).

CONTENTS

Report NM-R8512
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

TWO ADA MATHEMATICAL FUNCTIONS PACKAGES FOR USE IN REAL TIME

INTRODUCTION

This paper extends an investigation of the author, accomplished together with George T. Symm and Brian A. Wichmann of the National Physical Laboratory, Teddington, and Dik T. Winter of the Centrum voor Wiskunde en Informatica, Amsterdam, which resulted in a Final Report called "Guidelines for the design of large modular scientific libraries in Ada" (Symm et al. 1984, in the sequel referred to by "Guidelines").

Part of that report concentrated on the special requirements for library modules to be used as services by vital processes executing in a real-time environment. As an example, possible implementations are given here of a package of "basic mathematical function services" to be used by running processes.

1) Libraries for real-time use

For libraries to be used in a real-time processing environment, the special requirements are essentially that a running process (issuing a calculation request) cannot itself be interrupted, or can be kept waiting for only a limited (and probably very small) period (see Guidelines, Chapter 9, for more details).

For basic mathematical functions the time consumed by their calls is expected to be negligibly small. Nevertheless, the general recommendation given in the Guidelines ("services requested by tasks should always be granted by tasks") applies here too, because:

- it may be undesirable for the calling task to be suspended,
- perhaps the processor executing the calling task cannot perform floating-point operations,
- library modules for solving more complicated problems can request the same services.

We will restrict ourselves to those machine architectures for which the use of the Ada tasking concept makes sense. This is the case when multiple physical processors are available. With the tasking concept then actions to be performed in parallel can be described by several logical processors, and it is left to the Ada implementation how the logical processors are associated with the available physical processors.

We can illustrate what is possible by the following examples.

- We can imagine that the processors have their own (limited) storage resources; this may impose severe restrictions upon the storage size of a task's local declarations.
- Restrictions may also exist for the amount of use that can be made of memory accessible by all tasks. This affects the use of global declarations as well as the size of the channels the system will use for communication between tasks.
- We can even imagine that different physical processors have different floating-point characteristics. In that case tasks which require floating-point arithmetic can only be assigned to a subset of processors with the same floating-point representation and arithmetic, since all information through attributes and from the package SYSTEM is static. The user has no task placement control,

but he can expect that the implementation will associate his tasks only with the appropriate physical processors.
Especially with task types, one should be careful with the demand for local storage and with the use of shared variables, because several active tasks can then be created for which the scheduling may become complicated.
For the remainder of this paper we will assume that all demands for resources (for local declarations, for the use of globals, and for communication between tasks) are not excessive.

The implementations given below differ from the ordinary specification of a package (GENERIC_)MATH_FUNCTIONS (for sequential processing) in that (to the user) each function call is replaced by an entry call for a calculation REQUEST, followed by a call for the calculation COLLECT (these two can be integrated into one call if the answer can be "immediate").

It is still possible, however, to give a package specification identical to the specification of the package for sequential use (GENERIC_)MATH_FUNCTIONS (see Guidelines, Chapter 4) with function calls for the requested services. This is shown in the last section by a new package which encapsulates the earlier implementations with tasks. Extra costs for passing results should be small.

2) The package of server tasks

For the specification of a package of basic mathematical function services we propose two alternatives, the first allowing direct communication between customer and server for the collection of the result, the second with a "mailbox" construct for depositing the answer.
Each package is generic with respect to the type REAL to enhance its portability (see Guidelines, Chapters 3 and 4).

a. A package of servers with direct collection of results

The specification of the first alternative reads:

```
-------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTION_SERVERS is
   ----------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   ----------------------------------------------------------------
   -- Two typical server task declarations:

   task SQRT is

      entry REQUEST(X : in REAL);          -- to order a calculation
      entry COLLECT(RESULT : out REAL);    -- to obtain the result
      entry CANCEL;                        -- to cancel the order

   end SQRT;
```

```
   task LOG is

      entry REQUEST(X : in REAL; BASE : in REAL := EXP_1);
                                          -- to order a calculation
      entry COLLECT(RESULT : out REAL);   -- to obtain the result
      entry CANCEL;                       -- to cancel the order

   end LOG;


   -- All other basic functions analogous.
   ---------------------------------------------------------------
   -- No exception.
   ---------------------------------------------------------------
end GENERIC_MATH_FUNCTION_SERVERS;          -- specification
---------------------------------------------------------------
```

In a frame where an instance of the above generic package is mentioned in a use clause, services are obtained by simply issuing calls like SQRT.REQUEST(value); followed by SQRT.COLLECT(result);. Although the response time for this function will be negligible, the example can serve as a model for the implementation and use of larger services. If the calling task cannot wait any longer for an answer, it can decide to cancel in the following way:

```
SQRT.REQUEST(IN_VALUE);
OTHER_ACTIONS; -- by the calling task, finished after a certain
               -- time, or using a delay statement if more time
               -- is permitted to the server task.
select    -- a conditional entry call:
   SQRT.COLLECT(RESULT);
else
   SQRT.CANCEL;
   ALTERNATIVE_COMPUTATION;
end select;
```

b. A package of servers with mailboxes for depositing answers

The former implementation suffers from the fact that the server cannot be used again for a new service before the customer calls either COLLECT or CANCEL.

A different implementation is by using an "agent" task, usually called a "mailbox", which can receive a result (or in other examples even a succession of results) from the server task and which can be inspected by the calling task whenever necessary. See Guidelines, Chapter 9, section e.iii for full details.

The specification of the mailbox alternative is given below. We note in advance that (contrary to the example in the Guidelines) we have chosen an implementation in which the server (instead of the customer) creates the mailbox (through dynamic allocation).

```
-------------------------------------------------------------------
generic
   type REAL is digits <>;
package GENERIC_MATH_FUNCTION_SERVERS is
   -------------------------------------------------------------------
   PI : constant := 3.1415_92653_58979_32384_62643_38327_95029;
   EXP_1 : constant := 2.7182_81828_45904_52353_60287_47135_26625;
   -------------------------------------------------------------------

   -- Declaration of MAILBOX type:

   task type MAILBOX is
      entry DEPOSIT(X : in REAL);
      entry COLLECT(X : out REAL);
      entry CANCEL;
   end MAILBOX;  -- specification

   type ADDRESS is access MAILBOX;

   -- Two typical server task declarations:

   task SQRT is

      entry REQUEST(A : out ADDRESS; X : in REAL);
                                          -- to order a calculation
   end SQRT;


   task LOG is

      entry REQUEST(A : out ADDRESS; X : in REAL;
                BASE : in REAL := EXP_1); -- to order a calculation

   end LOG;


   -- All other basic functions analogous.
   -------------------------------------------------------------------
   -- No exception.
   -------------------------------------------------------------------
end GENERIC_MATH_FUNCTION_SERVERS;          -- specification
-------------------------------------------------------------------
```

A use of SQRT can be implemented as follows (assuming an instance of the above package to be mentioned in a use clause):

```
task USER;  -- specification

task body USER is
   SQRT_BOX : ADDRESS;
   --  ...
begin
   --  ...
   SQRT.REQUEST(SQRT_BOX, ARGUMENT);

   select
      SQRT_BOX.COLLECT(RESULT);
   or
      delay ALLOWED_TIME;
```

```
      SQRT_BOX.CANCEL;
      ALTERNATIVE_CALCULATION;
   end select;
   SQRT_BOX := null;

   -- ...
end USER;  -- body
```

The specifications have been chosen such that the package given in the Guidelines, Chapter 4, can be used to provide the implementations needed for all functions, as is shown in the next section.

3) Package bodies

We present here straightforward implementations for the servers defined in the previous section. For completeness, the package bodies each make their own instance of the GENERIC_MATH_FUNCTIONS package, although in an efficient implementation the bodies of the basic mathematical functions might be obtained in a different way.

We note that a server task cannot raise an exception like ARGUMENT_ERROR, because it is always TASKING_ERROR that is propagated to the customer. Therefore, this exception is omitted. The present implementation, where the value 0.0 is delivered in such circumstances, has its own drawbacks, but it is beyond the scope of this paper to offer a balanced recommendation.

a. Body for the "direct collection" alternative

In the following implementation the task body given (SQRT) contains an eternal loop in order to handle requests as long as needed, in the order the requests are received. The task receiving a request for a calculation will wait with the answer until the calling task collects the answer or issues a cancel of the request. It can then serve a new customer.
The bodies of all tasks look alike, except for the contained call of the function required. The calculation part might eventually be replaced by the appropriate sequence of statements.

```
with GENERIC_MATH_FUNCTIONS;
package body GENERIC_MATH_FUNCTION_SERVERS is

   package MATHFU_INSTANCE is
      new GENERIC_MATH_FUNCTIONS(REAL);

   task body SQRT is
      LOC_X, LOC_RESULT : REAL;
   begin
      loop
         select  -- for every service request
            accept REQUEST(X : in REAL) do
               LOC_X := X;
            end REQUEST; -- end of first rendezvous

            FUNCTION_CALL :
               begin
                  LOC_RESULT := MATHFU_INSTANCE.SQRT(LOC_X);
```

```
                exception
                   when MATHFU_INSTANCE.ARGUMENT_ERROR =>
                         LOC_RESULT := 0.0;
                end FUNCTION_CALL;

             select
                accept COLLECT(RESULT : out REAL) do
                   RESULT := LOC_RESULT;
                end COLLECT;
             or
                accept CANCEL;
             or
                terminate;
             end select;

          or
             terminate;
          end select;
       end loop;
    end SQRT;          -- body

    -- Other bodies analogous.

 end GENERIC_MATH_FUNCTION_SERVERS;
 ------------------------------------------------------------------
```

We note that if the customer task calls CANCEL the call will have to wait to be accepted until the server is ready, by when the task might as well collect the answer. Rewriting the code such that CANCEL calls can be accepted during the calculation requires a local task to perform this calculation, and the implementation would be more complicated than the version with a mailbox. It is more practical to omit the CANCEL entry here. An acceptable alternative is, that the CANCEL entry is polled at fixed stages during a long computation.

b. Implementation with "mailbox" construct

In the implementation using the mailbox the task SQRT receiving a request for a calculation will deposit the answer in the mailbox. It can then immediately serve a new customer. The customer may collect the answer whenever he wants, or otherwise cancel the mailbox, but only if the answer cannot be collected. It can be perceived that a CANCEL will also be handled by the mailbox after the acceptance of a DEPOSIT, because these entry calls may be issued at the same moment. If a CANCEL is handled first (the customer must not be kept waiting) then a DEPOSIT should yet be accepted.
It is recommended that the customer should not wait too long to collect his answer, since this would keep the mailbox from terminating.

```
-----------------------------------------------------------------------
with GENERIC_MATH_FUNCTIONS;
package body GENERIC_MATH_FUNCTION_SERVERS is

   package MATHFU_INSTANCE is
      new GENERIC_MATH_FUNCTIONS(REAL);

   task body MAILBOX is
      LOCAL : REAL;
   begin
      select
         accept DEPOSIT(X : in REAL) do
            LOCAL := X;
         end DEPOSIT;
         select
            accept COLLECT(X : out REAL) do
               X := LOCAL;
            end COLLECT;
         or
            accept CANCEL;
         or
            terminate;  -- in case the Customer has died.
         end select;
      or
         accept CANCEL;
         accept DEPOSIT(X : in REAL);
      end select;
   end MAILBOX;


   task body SQRT is
      LOC_X, LOC_RESULT : REAL;
      REPLY : ADDRESS;
   begin
      loop
         select  -- for every service request
            accept REQUEST(A : out ADDRESS; X : in REAL) do
               LOC_X := X;
               REPLY := new MAILBOX;
               A := REPLY;
            end REQUEST; -- end of first rendezvous

            begin
               LOC_RESULT := MATHFU_INSTANCE.SQRT (LOC_X);
            exception
               when MATHFU_INSTANCE.ARGUMENT_ERROR =>
                     LOC_RESULT := 0.0;
            end;

            REPLY.DEPOSIT(LOC_RESULT);
            REPLY := null; -- To allow the Mailbox to terminate.
         or
            terminate;
         end select;
      end loop;
   end SQRT;        -- body

   -- Other bodies analogous.
```

```
end GENERIC_MATH_FUNCTION_SERVERS;
```

---

4) Specification with function calls

It is possible to provide a package of basic mathematical functions whose specification is identical to that of the GENERIC_MATH_FUNCTIONS package (Guidelines, Chapter 4), but which can be used when the basic mathematical functions are available on a chip (a VLSI). This VLSI might satisfy one of the specifications given in section 2. above; a body is not needed. Every function body in the surrounding package body would consist of an entry call for the appropriate task. The use of tasks reflects the fact that the calculations are performed elsewhere.

As an example we give below a segment of the declarative part of such a package body using the "direct collection" version:

```
package MATH_FUNCTION_SERVERS is
   new GENERIC_MATH_FUNCTION_SERVERS(REAL);

function SQRT (X : REAL) return REAL is
   LOC_RES : REAL;
begin
   if X < 0.0 then raise ARGUMENT_ERROR; end if;
   MATH_FUNCTION_SERVERS.SQRT.REQUEST ( X );
   MATH_FUNCTION_SERVERS.SQRT.COLLECT ( LOC_RES );
   return LOC_RES;
end SQRT;
```

EPILOGUE

An interesting aspect of the above exercise was that the examples given in the Guidelines Chapter 9(e) concerning the "customer - server communication" had to be adapted before use. This was partly caused by the fact that in the application in this paper the mailbox was not used for depositing a series of gradually improving answers, but only for depositing one answer. It was then discovered that the implementations contained "busy waits". Moreover, the "mailbox" example supposed that both the customer and the server would tell the mailbox that they would not make further entry calls. We now consider this to be too much to ask of the customer, and any implementation should require no more communication between customer and "mailbox" task than strictly needed.

Yet another approach to supplying mathematical services is possible, with services like SQRT implemented as task types. The user can then communicate directly with his own server task, and mailboxes are not needed. The implementation (omitted here) is straightforward. The advantages to the user are similar to those of the mailbox approach, however, the task type construct might be easier for the task scheduler. Care must be taken that different objects of the same task type do not share global declarations which would cause them to wait for each other (we do not suppose that they would dare to update shared variables). A disadvantage is that the use of task types for the dynamic allocation of server tasks might eventually result in a shortage of storage. This is caused by the (expected) absence of a

Garbage Collector. It might therefore be wiser to re-use finished servers tasks, and the unauthorized use of a server might be prevented by issuing passwords to successive customers.

It can be expected that in the implementation of large scientific tools for use in real-time processes, many difficulties and surprises will yet arise. Use of the Ada concepts concerning "tasking" was found to be of great advantage for unveiling the difficulties indicated and for designing better solutions. We conclude that the effort needed to construct high quality libraries to be used in a real-time processing environment will turn out to be much larger than usually expected.

ACKNOWLEDGEMENTS

This paper has been endorsed by the Ada-Europe Numerics Working Group. The author is indebted to the members of the Working Group for the stimulating discussions, and to George Symm (NPL) for his editorial comments.

REFERENCE

Symm, G.T., Wichmann, B.A., Kok, J., and Winter, D.T. Guidelines for the design of large modular scientific libraries in Ada. Final Report, NPL Report DITC 37/84, Teddington, and CWI Report NM-N8401, Amsterdam, March 1984.