DAA/ AMES

Center for Reliable Computing

THE WATCHDOG TASK: CONCURRENT ERROR DETECTION USING ASSERTIONS

Aydin Ersoz, Dorothy M. Andrews, and Edward J. McCluskey

CRC Technical Report No. 85-8
(CSL TR No. 85-267)

July 1985

CENTER FOR RELIABLE COMPUTING
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305

Imprimatur: Aamer Mahmood and Saied Bozorgui-Nesbat

This work was supported in part by the NASA-AMES under contract No. NAG 2-246 and by the National Science Foundation under Grant No. DCR-8200129.

THE WATCHDOG TASK: CONCURRENT ERROR DETECTION USING ASSERTIONS

Aydin Ersoz, D. M. Andrews and E. J. McCluskey

CRC Technical Report No 85-8
(CSL TR No. 85-267)

May 1985

Center for Reliable Computing
Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305 USA

ABSTRACT

The Watchdog Task is a software abstraction of the Watchdog-processor. In this paper, the Watchdog Task is shown to be a powerful error detection tool with a great deal of flexibility and the advantages of watchdog techniques. A Watchdog Task system in Ada is presented; issues of recovery, latency, efficiency (communication) and preprocessing are discussed. Different applications, one of which is error detection on a single processor, are examined.

KEYWORDS: Concurrent checking, system level checking, watchdog, assertion, Ada[r], multiprocessing.

TABLE OF CONTENTS

LIST OF FIGURES

1 BACKGROUND

A Watchdog processor [Lu 80] is a small and simple coprocessor used to perform concurrent system-level error detection by monitoring the behaviour of a system (Fig. 1.1). The Watchdog is provided with specifications of the desired behaviour of the system before execution. During execution it compares concurrently collected information about the actual behaviour of the system with these specifications. A mismatch denotes an error.

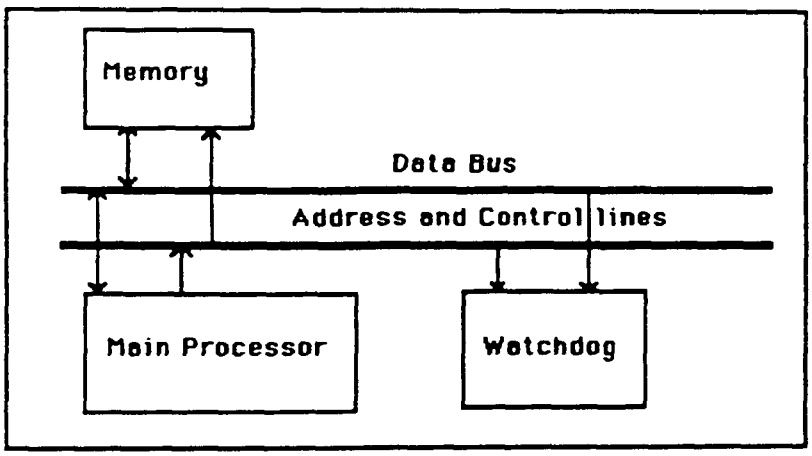

Fig. 1.1 The Organization of the Watchdog Processor System

Many implementations of the Watchdog processor are discussed in the literature. The distinctive characteristic of a Watchdog is the particular aspect of the system behaviour it monitors. Watchdogs have been proposed that monitor the memory access behaviour [Namjoo 82a], the control flow [Lu 82] [Namjoo 82b] [Sridhar 82] [Namjoo 83] [Shen

83] [Eifert 84], the control signals [Daniels 83] and the reasonableness of results [Saib 79][Mahmood 83].

For details on the Watchdog Processor, the reader is referred to [Mahmood 85], an excellent survey from which most of section 1. has been taken.

Assertions and the Assertion Watchdog Processor:

The scheme based on checking the reasonableness of results [Mahmood 83] relies on assertions inserted into the program running on the main processor. An assertion is an invariant relationship about the variables of a program written as a logical statement and inserted at different points in the program. It signifies what is believed to be true at the point the assertion is inserted. The following examples of assertion has been taken from the code for a flight software package [Mahmood 83]:

```
Define Procedure LAT.INNER to be
 begin
 .
 if R.TEST.COMPL then begin
    RL8 = RL8.D = DLIMIT (RL8.D+RL11.D+RL11.D.S,0.258);
    RL11.D.S = RL11.D;
    RL13 = LIMIT(RL7+RL8,0.171429)/0.20333;
 end
 else RL3 = TEST.CMD (RAM.PTR (R.TEST.PTR));
 .
 DELA.CMD = RL13;
COMMENT ASSERT ABS(DELA.CMD) < 0.13;
 .
 end;
```

Anna, a specification language for Ada, provides comprehensive support for assertions [Luckham 84]. The following example is a result annotation defining the value returned by a function:

```
function COMMON_PRIME (M,N : INTEGER) return NATURAL;
--| where return P : NATURAL =>
--|    IS_PRIME(P) and M mod P =  0 and N mod P = 0;
```

The use of assertions provides the notion of correctness that is necessary to do semantic (reasonabless of results) checking. The effectiveness of assertions in detecting all types of errors (hardware as well as software) has been demonstrated in [Andrews 81] and [Mahmood 84].

Assertions could certainly be executed on the same processor as the program they check, but dedicating a Watchdog processor to this task has several advantages:

1. Efficiency : The assertions are executed concurrently on a different processor. This is particularly valuable in real time systems, especially if the assertions are inserted after the system has been developed. If the execution overhead of the assertions has not been taken into consideration in the initial design, it is likely that their effect on the timing will be intolerable if executed on the same processor as the main program.
2. With a Watchdog processor, the checking is done by an independent module.This independency is always desirable in testing.
3. The dedicated Watchdog processor can be specially designed to execute the assertions efficiently.

In a system that employs an Assertion Watchdog, the 'main' program running on the system is preprocessed to extract the assertions. These are then arranged to form the code for the Watchdog

processor. In a general implementation of this scheme [Mahmood 85], the Main processor has to make the data necessary for the execution of the assertions available to the Watchdog. This explicit communication constitutes the bottleneck in the efficiency of such a Watchdog system.

2 A MULTIPROCESSING MODEL OF THE ASSERTION WATCHDOG PROCESSOR SYSTEM

The system described in the preceding section can be modelled as two communicating concurrent processes. As depicted in Figure 2.1, process Main is the program running on the Main processor, the other process is the Watchdog, which is made up of the assertions. The communication consists of the data for the assertions that process Main transfers to process Watchdog.

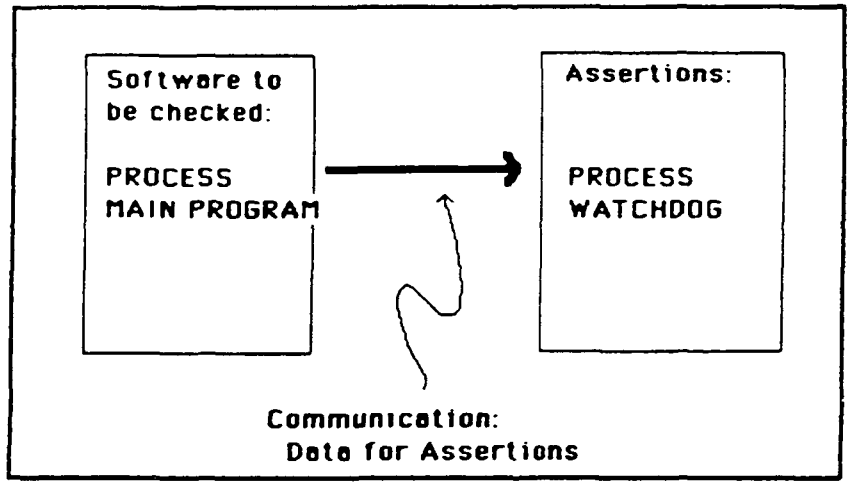

Fig. 2.1 The Multiprocessing Model of the Assertion Watchdog System

The multiprocessing model is abstract in the sense that it is not bound to a particular implementation. The Assertion Watchdog monitors

the semantics of the Main process; the communication between the processes is the explicit transfer of the values of program variables. This independence of the Assertion Watchdog from low level implementation issues is in contrast to the other watchdogs. Watchdogs that perform structural integrity checking are also high level in the type of checking they perform, and can be described as abstract multiprocessing models [Lu 82]. With the Assertion Watchdog, however; the processes are the actual programs running on the processors, and the communication is program variables as opposed to computed labels. The control-flow watchdog [Namjoo 82b] for example, involves monitoring and compressing the instruction flow of the main processor. The assertion Watchdog system, when viewed as two concurrent processes, can be conceptualized without any reference to the actual hardware. It would be possible to schedule the two processes that make up the system on any single or multiprocessor architecture.

3 THE WATCHDOG TASK

The Watchdog Processor is a powerful error-detection tool that is completely transparent to the software running on the system. The Watchdog Task is an analogous tool exclusively at the software level. A software system with a Watchdog process can be built in a multiprocessing environment without paying attention to the details of the physical architecture. The abstract nature of the Assertion Watchdog is the key to the feasibility of this scheme.

Such a system can be implemented in a High Level Language (HLL) that supports multiprocessing. The language should provide constructs for expressing the concurrency and communication which are the very essence of the Watchdog system.

The system that is presented in this paper is written in Ada. In Ada terminology, processes are called tasks; hence the title 'The Watchdog Task'.

Using the Software Watchdog:

The Software Watchdog can be used in developing a reliable software system as follows: The starting point is a sequential ADA program with assertions already inserted in it. A preprocessor transforms this Main program into a Watchdog Task System with multiple concurrent tasks, some of which are watchdogs. This system, which is in ADA, can then be compiled and the different tasks scheduled on the actual hardware.

4 AN EVALUATION OF THE SOFTWARE WATCHDOG CONCEPT

Now that the feasibility of the Software Watchdog system has been described, it is worthwhile to examine its strengths and the difficulties involved with its implementation before presenting the actual system. In this section, it will be demonstrated that this high level language approach to concurrent error detection generalizes the watchdog concept and simplifies its integration into a system.

4.1 Advantages:

The arguments for using a HLL in any application apply to the watchdog case as well.

1. Flexibility: The HLL approach results in a great deal of flexibility. The system can now be conceptualized in the most convenient way. It is no longer necessary, for example, to confine the system to one watchdog; many Watchdog Tasks may be scheduled. It will be shown later how such conceptualizations prove to be valuable.

2. Tools for Handling Recovery and Latency: In a system with error detection, it is usually desirable to recover from the errors in some specified fashion. The recovery procedure can be built into the software watchdog, resulting in the integration of a reliable system. The software system also provides the user with tools for handling the latency problem, as discussed later (section 6.3).

3. Portability: The Watchdog Task System that is derived from a Main program is in ADA and is almost completely independent of architecture. Except for one package that contains hardware information (see sections 4.3 and 6.4.1), the system is portable and can run on any single or multiprocessor hardware with an Ada code generator.

4. Single Processor Architectures: A New Application for the Watchdog: The generality of the software watchdog makes it possible to run the system without any changes on a single processor,

combining the watchdog and background checking techniques in a natural way.

5. Simulation: The implementation of a software watchdog is clearly simpler than implementing a hardware system with a watchdog processor. The software system can therefore be utilized for simulating and evaluating different watchdog schemes (particularly the communication aspect).

4.2 Preprocessing:

Preprocessing is required in any system utilizing a watchdog processor. In the case of the Assertion Watchdog, the assertions have to be extracted from the original code and put together to create the program for the watchdog processor.

If a software watchdog is built, no further preprocessing is necessary since the Watchdog Task is precisely the code for the watchdog processor. The effort that goes into transforming a Main program into a Watchdog Task system, in other words, is not extra overhead; it simply replaces the previous preprocessing.

4.3 Difficulties Associated with Implementation:

It was discussed by way of background (Sec. 1.1) that a good part of the motivation behind the watchdog was an improvement in efficiency. It is certainly important that the software watchdog detracts minimally from the efficiency of the underlying system. There are two conflicting goals here: The software watchdog is implemented in a HLL and is meant to be an abstraction that has little

dependence on the architecture. However; in order to maximize efficiency, it is desirable to have an accurate model of the architecture to exploit its potential fully.

The only aspect of the architecture that has a major impact on efficiency is the implementation of the communication between processes Main and Watchdog. It is therefore possible to isolate the critical hardware and map it into software. The intersection of the software abstraction and the actual hardware, then, is well defined and very limited (Fig 4.1). The hardware dependency can be expressed in an Ada package which needs to be written before the software watchdog can be installed on a hardware system. (section 6.4.1). This limited hardware map constitutes a satisfactory compromise between abstraction and full exploitation of the hardware.

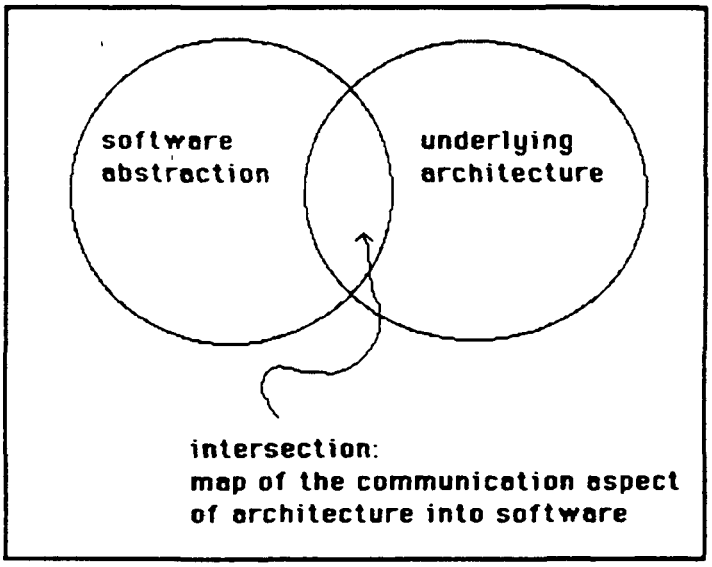

Fig. 4.1 The Architectural Dependency

The HLL implementation also gives rise to a quite different type of efficiency problem. Many HLL constructs such as procedure calls

result in additional execution overhead. Clearly, this overhead can be controlled by paying attention to the runtime behaviour of the code when implementing the Watchdog System.

5 DEFINITION OF THE RESEARCH PROBLEM

The preprocessing that transforms the sequential Main program into a Watchdog Task system is uniform. Different watchdog systems will differ in the main program code and assertions they contain, but will have the same basic structure. The objective of the research presented in this paper is to design this underlying structure. Once the basic format of the Watchdog Task system is determined, the transformation from sequential Main program to system will be straightforward.

The system template is designed with the following criteria in mind:

1. The system should be general enough that any sequential main program can be transformed into a Watchdog Task system through simple preprocessing.
2. It should provide tools for controlling recovery and latency.
3. It should be 'efficient'. The notion of efficiency will be made explicit in section 6.4.

6 THE WATCHDOG TASK SYSTEM

In this section, the system that was designed will be presented. It should be emphasized that this system was meant to be completely general. Improvements would be possible if the software is custom designed for an application (section 8).

The essential structure of the system will be discussed first; the recovery and communication mechanisms will be covered later. The format will be illustrated by actual examples from a flight software package that was rewritten in Ada.

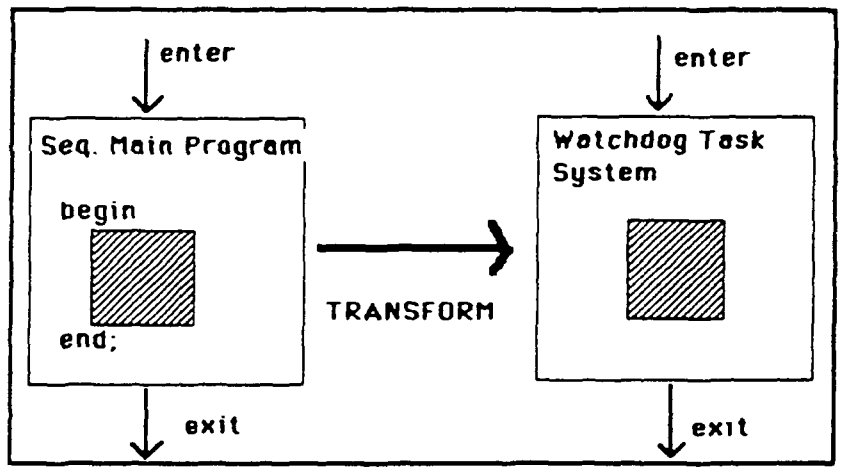

Fig. 6.1 The Transparency of Transformation

6.1 The Essential Structure:

The Watchdog Task system is a multitasking Ada program which can be activated from the external environment via a START procedure. The

transformation of the main program to a multitasking system is transparent to the caller of the main program. The main program is replaced by a call to procedure START which returns control to the caller after the main program code and the assertions have been executed. Figure 6.1 shows the transparency of the transformation.

The system consists of two packages, which are outlined below:

```
package SUPERVISOR is
     NUMBER_OF_ASSERTIONS : constant := 19;
     type ERROR_RANGE is range 1..NUMBER_OF_ASSERTIONS;
     procedure START;
     procedure REPORT (ERROR : ERROR_RANGE);
     function IS_DONE return BOOLEAN;
end SUPERVISOR;

package body SUPERVISOR is

     DONE : BOOLEAN;

     task MAIN is
        entry START;
     end MAIN;

     task body MAIN is separate;

     procedure START is
        . . .
     end START;

     procedure REPORT(ERROR : ERROR_RANGE) is
        . . .
     end REPORT;

     function IS_DONE return BOOLEAN is
        begin
            return DONE;
        end IS_DONE;

begin
     DONE := FALSE;
end SUPERVISOR;
```

```
with SUPERVISOR;
package WATCHDOG is
     function IS_IDLE return BOOLEAN;
     package QUEUE is
     .
     end QUEUE;
end WATCHDOG;

package body WATCHDOG is

     package body QUEUE is separate;

     function IS_IDLE return BOOLEAN is
        .
     end IS_IDLE;

     task CHECK;
     task body CHECK is
        .
        begin
            if not SUPERVISOR.IS_DONE then
                --execute the assertions
                .
        end CHECK;
end WATCHDOG;
```

Package SUPERVISOR contains the system's interface with the external environment and the main program code. Package WATCHDOG contains a communications package which will be explained in section 6.4.1, and the code for executing the assertions. The main program code is in task SUPERVISOR.MAIN, its internal structure (such as local procedures, etc.) has become local to task MAIN.

```
with WATCHDOG;
separate(SUPERVISOR)
    task body MAIN is
       -- local declarations, subprograms, packages, etc. are inserted
       -- here

       procedure WAIT_FOR_WATCHDOG is
           begin
               WHILE not WATCHDOG.IS_IDLE loop
                  delay WAIT_TIME;
               end loop;
           end WAIT_FOR_WATCHDOG;
```

```
begin
   accept START do
      -- the sequential main program is inserted here
          .
      WAIT_FOR_WATCHDOG;
   end START;
end MAIN;
```

In order to control the execution of task MAIN, its body is inserted within an accept statement. A call to MAIN.START starts the main program, the caller gets control back only when all of MAIN has been executed. The net effect is that MAIN gets executed sequentially with the caller. This is the desired behaviour for the transparency of the system.

The assertions are executed within the body of task WATCHDOG.CHECK. This task has to be compiled separately (that is why it is in a separate package) for full generality. The main program may be compiled with different units and it may be desirable to call the watchdog from within these units. If the WATCHDOG package is separate, compiling any unit that needs the watchdog with package WATCHDOG establishes the required visibility.

The watchdog executes in an infinite loop and terminates when the flag SUPERVISOR.IS_DONE is set. The way the body of the task is organized is mainly an artifact of communication requirements and will be discussed in connection with this issue (section 6.4.1). When the watchdog detects an error, it calls procedure SUPERVISOR.REPORT which aborts task MAIN. SUPERVISOR.REPORT also logs the error:

```
procedure REPORT (ERROR : ERROR_RANGE) is
   begin
      --log the error:
      case ERROR is
```

```
            when 1 => ...
            .
        end case;
        abort MAIN;
    end REPORT;
```

Procedure SUPERVISOR.START is what the external world uses to activate the system:

```
procedure START is

   begin
        MAIN.START;
        DONE := TRUE;  -- signal WATCHDOG to stop
   exception
        when TASKING_ERROR =>
            -- handle error reported by WATCHDOG;
        when others =>
            -- handle other exceptions;
   end START;
```

The procedure remains in rendezvous with task MAIN until either the task completes successfully, or is aborted. Upon normal completion, the watchdog is signalled to terminate through flag IS_DONE. If the task has been aborted, then the exception TASKING_ERROR is raised since a rendezvous was in progress. The exception is detected, and the recovery mechanism is initiated. This will be elaborated in section 6.2.

The essential structure of the Watchdog Task System is summarized in Figure 6.2. When procedure SUPERVISOR.START is called, it activates task MAIN. The watchdog remains active until MAIN completes execution, at which point SUPERVISOR.START is also exited.

As a last remark, notice that the system is guaranteed to have executed all assertions since task MAIN explicitly waits for the

watchdog to catch up by calling WAIT_FOR_WATCHDOG before completing. WAIT_FOR_WATCHDOG simply checks to make sure the watchdog is idle.

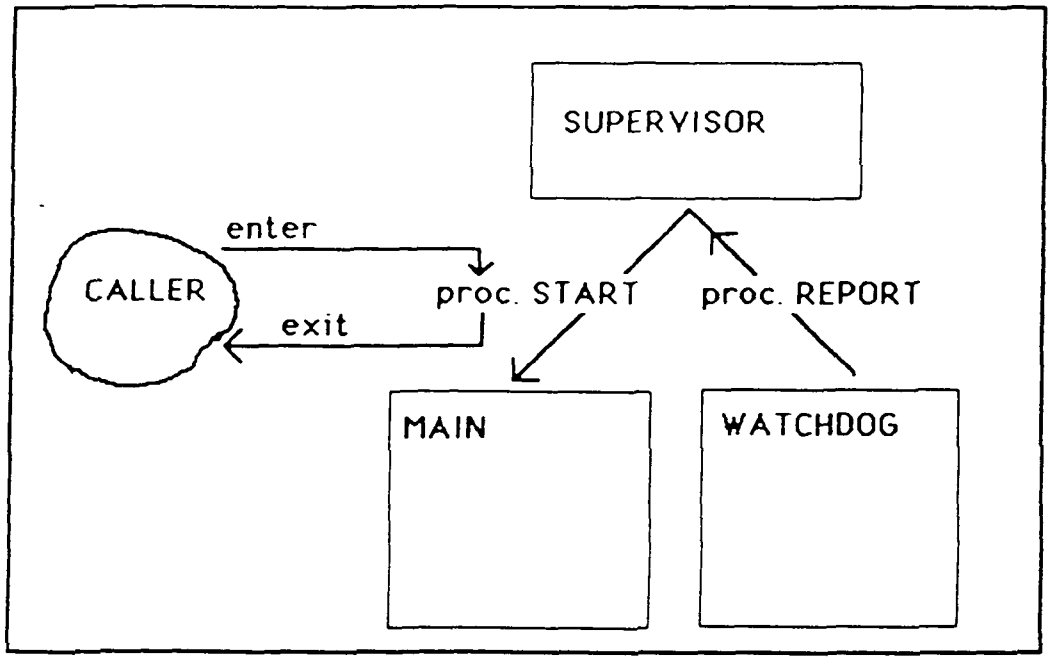

Fig. 6.2 The Basic Structure of the Watchdog Task System

The type visibility problem:

The watchdog and the main program are transformed into two separate tasks, but the watchdog task needs data from task MAIN. The so called type visibility problem arises because these data may be of types local to task MAIN; indeed, the types could be declared deep within the hierarchy of the main program. These local types need to be made visible to the watchdog task.

One solution to the problem would be to move the relevant local declarations to the global level and make them accessible to the watchdog. This is clearly not desirable since it violates the basic principles of structured programming. The alternate solution, the one

that was implemented in the Watchdog Task system, is to convert the data of local types into predefined ADA types. Data of composite types need to be broken down before the communication in any case, so the conversion is straightforward, especially if the domain is restricted to a small number of types. The conversion problem can be completely resolved at preprocessing time.

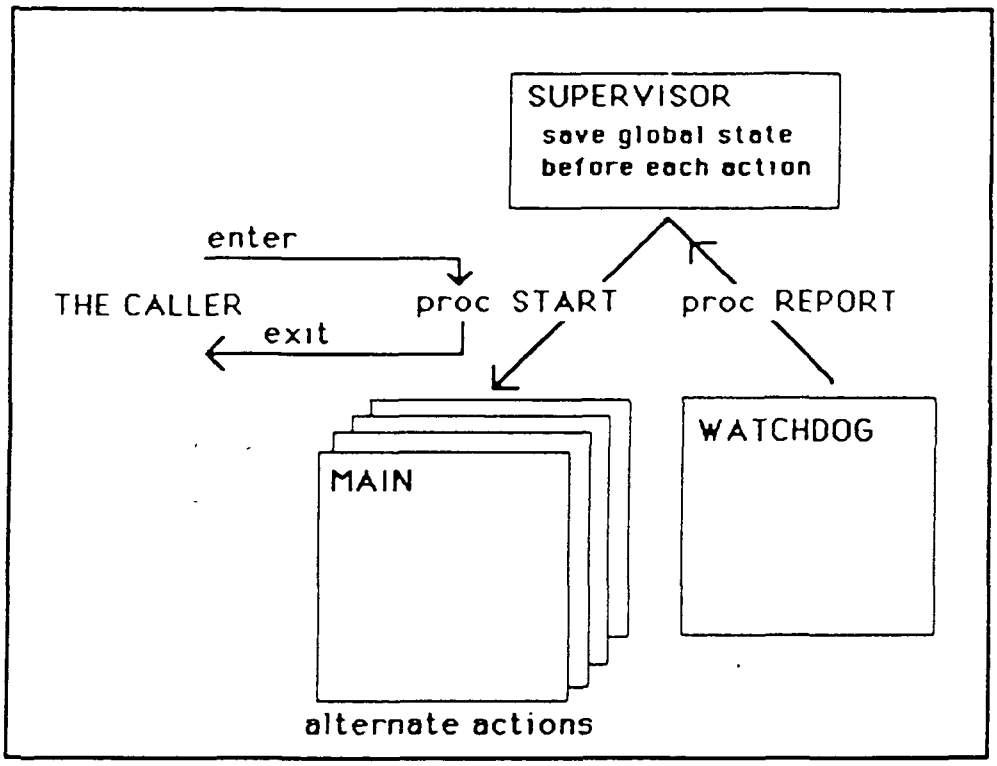

Fig. 6 3 The Recovery Mechanism

6.2 The Recovery Mechanism:

The standard recovery mechanism of having alternate actions available in case of failure [Randell 75] is integrated into the Watchdog Task system. Figure 6.3 describes the arrangement. The 'alternate actions' are different tasks that are internal to package

SUPERVISOR. These may be programs that are different implementations of the main program, or simple error handlers. The code for procedure SUPERVISOR.START is modified as follows to accomodate the recovery:

```
type ACTION is ...
procedure START is

   CURRENT_ACTION : ACTION;

   begin
        INITIALIZE_CURRENT_ACTION;
        loop
             begin
                 -- save global state;
                 case CURRENT_ACTION is
                     when ... => MAIN.START;
                     .
                     when ... => --handle error;
                 end case;
                 exit;
             exception
                 when TASKING_ERROR =>
                     UPDATE_CURRENT_ACTION;
                     -- restore global state;
                 when others =>
                     -- handle other exceptions;
             end;
        end loop;
        DONE := TRUE;  -- signal WATCHDOG to stop
   end START;
```

CURRENT_ACTION is a global variable that keeps track of the actions being performed. If the task that is the current action completes successfully, the watchdog is terminated and SUPERVISOR.START is exited. If the CURRENT_ACTION fails and is aborted, exception TASKING_ERROR is raised, CURRENT_ACTION is updated, and a new task is started. Notice that this mechanism may be used to handle other exceptions that propagate from task MAIN.

It may be necessary to 'roll back' the task that failed and was aborted. If this is desired, the global state must be saved before the task is activated and restored after it is aborted. This is

discussed in [Randell 75]. The preprocessing can determine the variables that need to be saved by scanning the code for places where they are modified. Since task Main is a functional unit, this scan is unlikely to be overly complicated.

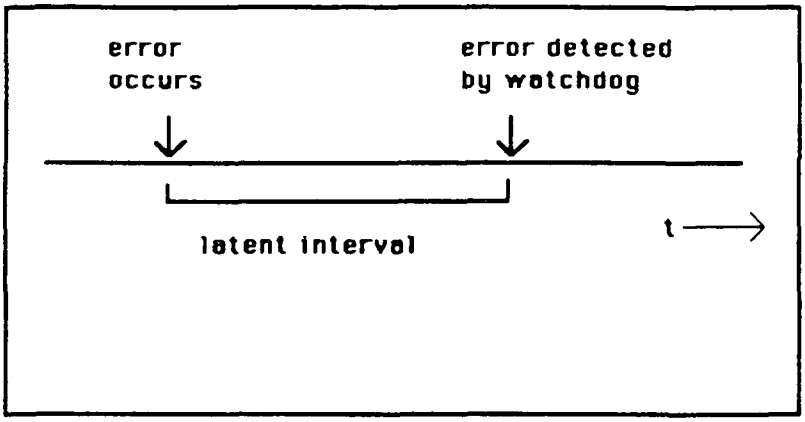

Fig. 6.4 The Latency Problem

6.3 The Latency Problem:

As shown in Figure 6.4, the watchdog is not guaranteed to detect errors precisely when they occur. This leads to the presence of a 'latent interval' from the time the error has occurred to the time it is detected, during which the system is no longer operating properly. This latency is intolerable if the system performs a critical action during the latent interval. The latency problem is a fundamental one and cannot be completely solved in the general case. Assertions and the software watchdog, however, provide a tool that brings latency under control by confirming the correctness of the program before executing a critical action.

In the following, it will be assumed that the assertions inserted into the program are complete in the sense that the correct evaluation

of all assertions inserted up to a critical point means that the program contains no errors up to that point. This will make it possible to isolate the latency of the watchdog from the latency of the assertions. With this assumption, all that is required of the watchdog system is to make sure there are no unexecuted assertions when the main program arrives at a critical point.

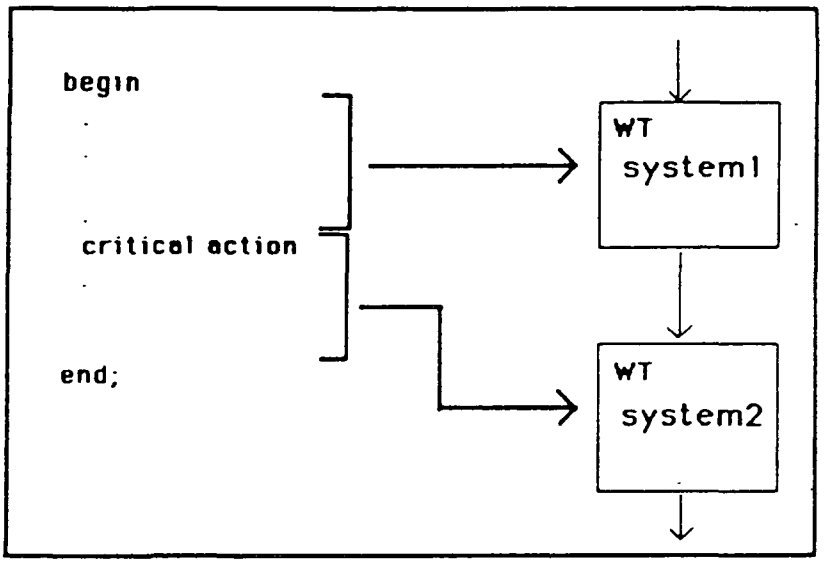

Fig. 6.5 Transforming the Program into Several Watchdogs to Control Latency

The solution is to suspend the execution of task MAIN until the watchdog catches up by waiting until WATCHDOG.IS_IDLE. An implementation of this mechanism will be presented in the next section. As explained in section 6.1, this wait is already built into the system so that all assertions are executed before the system can be exited. A more formal solution to the latency problem, therefore, is to break the sequential main program into subparts delimited by critical actions. A Watchdog Task system can then be generated for each as shown in Figure 6.5. This is a natural

formalization if the critical actions are relatively sparse and independent.

6.4. Efficiency- Interprocess Communication:

The question of efficiency is meaningful when the Watchdog Task system is to be scheduled on a multiprocessor. Efficiency, in the sense it is employed here, is related to the total time it takes the system to execute both the main program code and the assertions. The purpose of dedicating a processor to the watchdog process is to reduce this total time. As mentioned in section 4, it is critical that the software watchdog detracts as little as possible from the gain in efficiency achieved through this parallelism.

It is easy to demonstrate that the bottleneck in efficiency is communication. The following is the inequality that should be maximized by the Watchdog Task system:

Time of execution for main program + assertions > Time of execution for Watchdog Task System

If perfect parallelism were possible, the left hand side of the inequality would exceed the right hand side by 'Time of execution for assertions'.

The inequality transforms into:

time of execution for main program + assertions > max [time of execution for (task MAIN, watchdog task)]

Task MAIN will clearly take longer to execute than the watchdog since the assertions are typically a fraction of the original code and are being executed on a specialized processor. Also noting that

time of execution for task MAIN = time of execution for main program + time for data transfer to watchdog,

the inequality becomes:

time of execution for assertions > time for data transfer to watchdog.

Since the left hand side of the inequality is constant, the inequality can only be maximized by minimizing the transfer time spent by task MAIN. To maximize performance, therefore, the communication cost has to be minimized.

It is also interesting to note that this inequality is critically dependent on the type of assertions being executed. If the assertions are computationally intensive, a less than optimal communication scheme may be tolerable since the inequality will clearly be dominated by the execution time of the assertions. If, on the other hand, the assertions are data intensive, the communication cost will have to be kept very low to justify executing the assertions in parallel.

Implementing Communication:

The most straightforward implementation of intertask communication in Ada would be by rendezvous. Unfortunately, the rendezvous allows only for tight synchronous coupling between tasks and is unsuitable for the kind of asynchronous communication that is needed for the watchdog. In an implementation employing the rendezvous, task MAIN would have to wait for the watchdog task everytime it tried to send new data. In the general case, this could reduce the parallelism drastically and is clearly unacceptable.

Since the rendezvous construct cannot be utilized, the asynchronous intertask communication has to be constructed explicitly in software. In order to maximize the efficiency, it is necessary to model the communication that actually goes on between processors as accurately as possible.

At this point, then, the discussion becomes dependent on the organization of the underlying system. The implementation that will be presented assumes a two processor architecture and remains as general as possible within the bounds of this assumption. The dual processor architecture, with a main and a watchdog processor, is the general scheme proposed in the literature.

Two processors generally communicate via a buffer or queue as shown in Figure 6.6. The main processor only writes into the buffer, and the watchdog processor only reads from the buffer. The specification for the package that models this buffer is presented below.

```
package QUEUE is
   type DATA_TYPE is (INT, FL_POINT); --all  standard data types
                                      --used in assertions;
   type UNIT_DATA (KIND : DATA_TYPE := FL_POINT) is
       record
            case KIND is
                when INT =>
                   INT_ ATA : INTEGER;
                when FL_POINT =>
                   FL_DATA : FLOAT;
            end case;
       end record;

   MAX_LABEL : constant := 19;
   type LABEL_TYPE is range 1..MAX_LABEL; -- the type for the
                                          --assertion labels
   MAX_NUMBER_OF_PARS : constant := 10;   --the max. number of
                                         --data items for the
                                         --assertions
   type PAR_RANGE is range 1..MAX_NUMBER_OF_PARS;

   --DATA_ARRAY is defined to simplify getting data from the QUEUE
   type DATA_ARRAY is array (PAR_RANGE range <>) of UNIT_DATA;

   function IS_EMPTY return BOOLEAN;
   procedure REQUEST_SPACE (FOR_NEXT    :   PAR_RANGE);
   procedure INSERT         (LABEL      :   in LABEL_TYPE);
   procedure INSERT         (FL_DAtA    :   in FLOAT);
   procedure INSERT         (INT_DATA   :  in INTEGER);
   procedure GET_NEXT       (LABEL      :  out LABEL_TYPE);
   procedure GET_NEXT       (NO_OF_PARS  : in PAR_RANGE;
                            DATA        :  out DATA_ARRAY);
   procedure FLUSH;

   pragma INLINE(INSERT, GET_NEXT, IS_EMPTY, FLUSH,
                 REQUEST_SPACE)
end QUEUE;
```

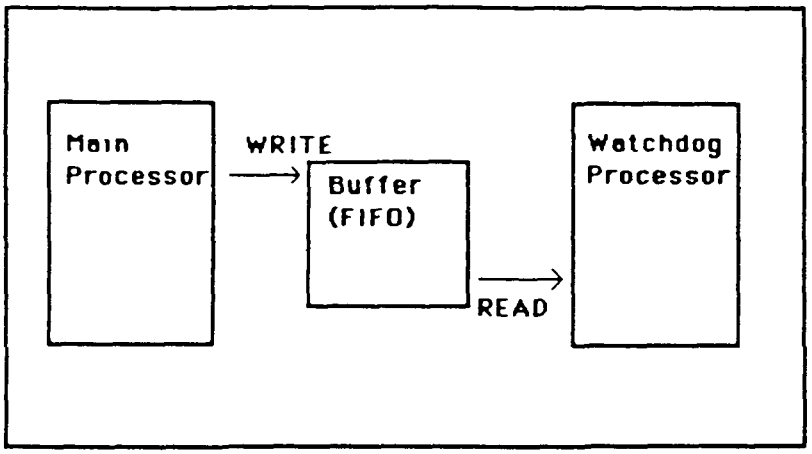

Fig. 6.6 Communication in Dual Processor Architecture

Package QUEUE is contained in package WATCHDOG. The specification contains the data types and the basic QUEUE operations and attributes. All procedures and functions are compiled with the pragma INLINE to avoid the procedure call overhead. The basic data item in the queue is a variant record which has fields for all the data types that will get transferred, including the type for the assertion labels.

Task MAIN enters the data into the queue by first requesting space for the data and then inserting it. The following example is from the flight software package.

```
ASSERTION_4:
begin
  WATCHDOG.REQUEST_SPACE(FOR_NEXT => 4);
  WATCHDOG.QUEUE.INSERT(LABEL => 4);
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(LAT_INN_CMD));
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(RL5));
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(ROLL));
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(ROLL_RATE));
end ASSERTION_4;
```

The REQUEST_SPACE procedure is a simple check to see if the queue is

full. If it is, task MAIN has to wait until the watchdog processes the next assertion from the queue, making more space available.

The watchdog side of the communication is equally straightforward.

```
 task body CHECK is
   LABEL : QUEUE.LABEL_TYPE;
begin
   loop
       if QUEUE.IS_EMPTY then
           if SUPERVISOR.IS_DONE then exit;
           else delay WAIT_TIME;
           end if;
       end if;
       while not QUEUE.IS_EMPTY loop
           QUEUE.GET_NEXT(LABEL);
           case LABEL is
               when 1 => ...
               .
               when 4 =>
                  ASSERT_4:
                  declare
                      PACKET : QUEUE.DATA_ARRAY(1..4);
                  begin
                      QUEUE.GET_NEXT(NO_OF_PARS => 4,
                                                   DATA => PACKET);
                      if abs(PACKET(1).FL_DATA -
                               0.5*
                               (PACKET(2).FL_DATA +
                                PACKET(3).FL_DATA * 0.764
                                PACKET(4).FL_DATA * 0.152533)) >
                          0.0001 then
                            SUPERVISOR.REPORT(ERROR => 4);
                            QUEUE.FLUSH;
                      end if;
                  end ASSERT_4;
           end case;
       end loop;
   end loop;
end CHECK;
```

If QUEUE.IS_EMPTY, there are no assertions to be executed and the watchdog is idle. When there is data in the queue, the watchdog reads the label for the next assertion, gets its data and processes it. Function WATCHDOG.IS_IDLE, which task MAIN uses to find out if the

watchdog is still executing assertions, is implemented as a simple boolean operation that checks if the queue is empty. If the Watchdog detects an error that will result in aborting task MAIN, it invalidates the data for the remaining assertions by flushing the queue.

The body of package QUEUE contains the actual hardware mapping in terms of the implementations of the queue operations. As explained in section 4.3, this isolates the hardware dependency. A different body for QUEUE needs to be written for the different buffering schemes.

The buffering schemes can cover a wide spectrum. The packages for a representative set have been written and simulated on a single processor system. These differ in the type of the queue items, the complexity of the queue management, and the assumptions made about the memory the queue resides in.

The items in the queue can either be a simple data item as in the package specification presented above (QUEUE. UNIT_DATA), or a packet of varying size that contains the label for the assertion and its data (Fig. 6.7a). The packet is clearly a more convenient representation to work with, but its varying size makes queue management considerably more complex. The packet representation would not be feasible unless it is supported by the underlying architecture and operating system. Examples of a linked list and ring buffer with packets (Fig. 6.7b) are given in the appendix. Again, these schemes would only be feasible if the additional cost of queue management is justified.

For the case where the underlying system provides no support other than a memory between two processors (Fig. 6.6), WATCHDOG.QUEUE has to be more 'low level'. The data items have to be inserted one by one

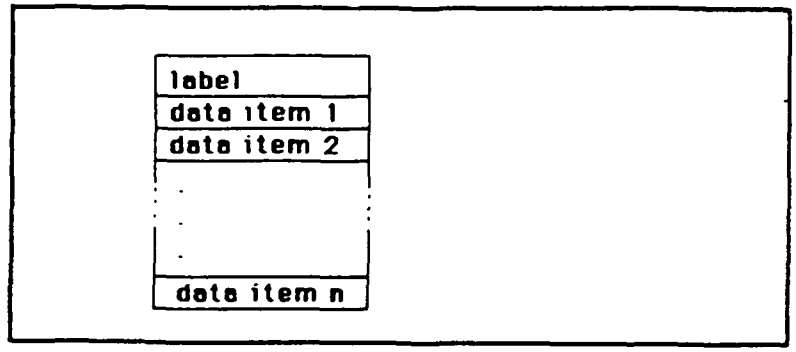

Fig. 6.7a The Packet

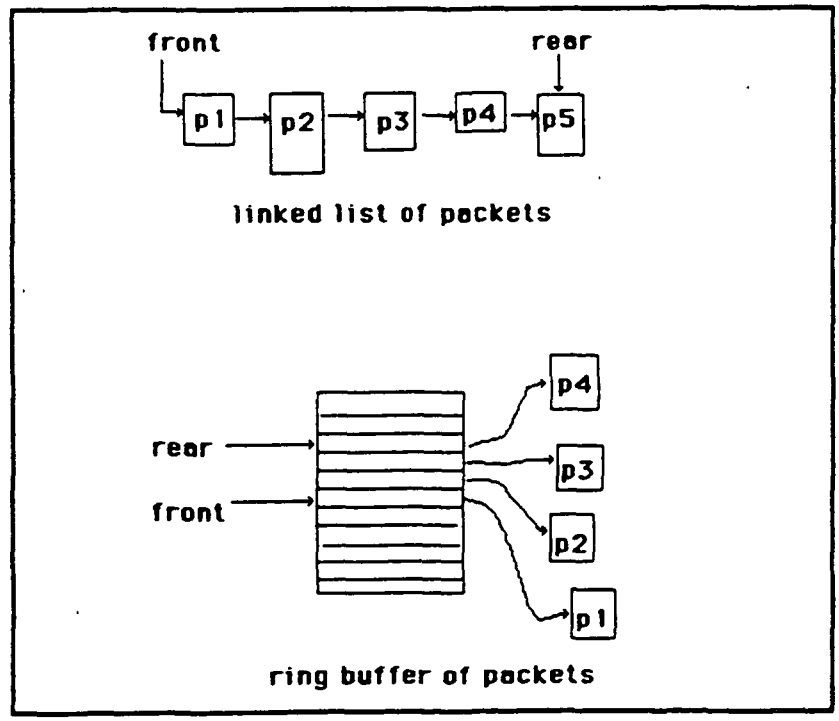

Fig. 6.7b Queues of Packets

(as opposed to a packet per assertion) and the queue management has to be limited to schemes no more complicated than a ring buffer for cost-effective communication. If the memory is dual ported (one write, one read port), a ring buffer of data items establishes the desired communication (Fig. 6.8). If the physical memory that is available is single ported, it needs to be converted into the equivalent of a dual ported memory by using a double-buffer scheme. The performance of the single ported memory is unacceptable since it leads to the same problem that came up with rendezvous communication: Task MAIN has to wait until the watchdog finishes accessing the memory.

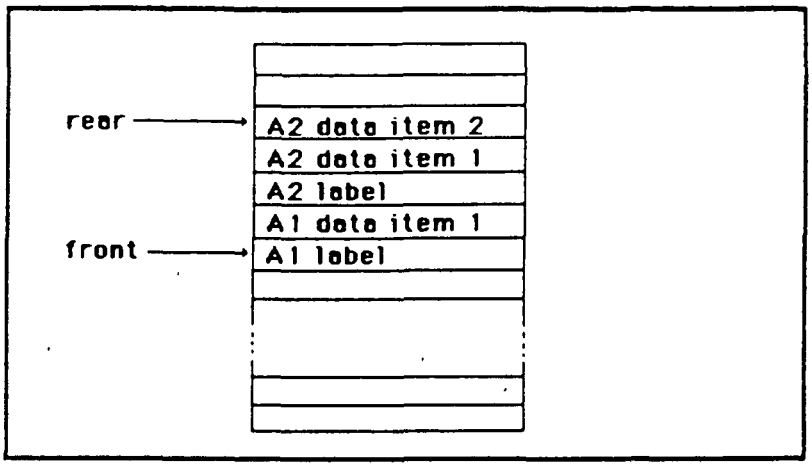

Fig. 6.8 Ring Buffer of Unit Size Data Items

A double-buffer is shown in Fig. 6.9. Task MAIN writes into one buffer and the watchdog reads from the other. The buffers are switched when the current buffer of the watchdog is empty and the current buffer of the main task is filled up over a specified level. The swapping time can be tuned to the system. The buffers no longer

need to be ring buffers; simple arrays are sufficient. The details of the packages for both of these schemes are given in the appendix.

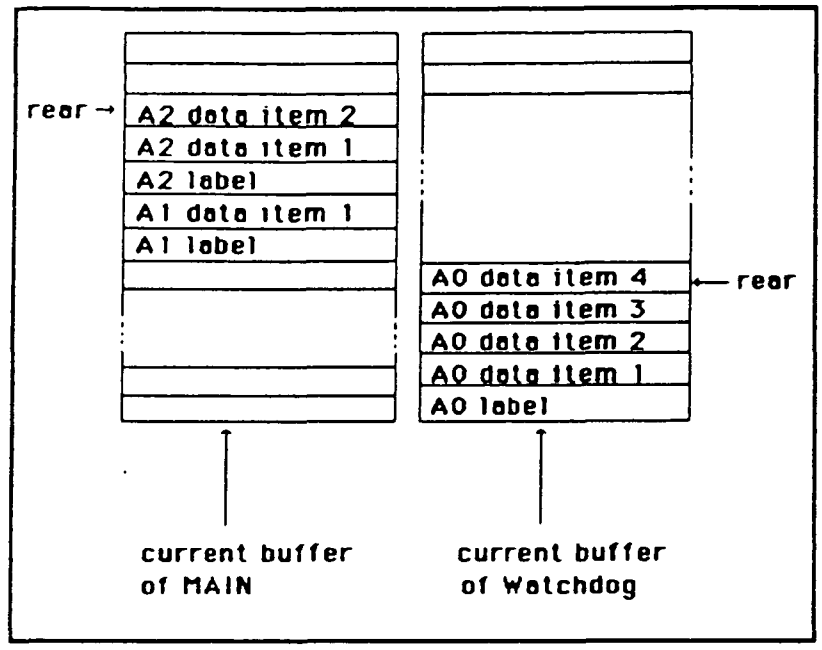

Fig. 6.9 The Double Buffer

All support provided by the underlying system for interprocess communication should be exploited in implementing the package WATCHDOG.QUEUE. The overhead for WATCHDOG.QUEUE.REQUEST_SPACE, for example, can easily be disposed of in many cases. If the buffer is large enough, or if the watchdog processor is fast enough, in short, if the buffer is guaranteed to never overflow, it will be unnecessary to check if it is full. Even if this is not the case, some simple logic built into the physical buffer will be sufficient to avoid QUEUE.REQUEST_SPACE. Since the assertions are preprocessed, the size of the largest data packet is known. A simple queue manager monitoring the buffer can use this information to check if the buffer is 'dangerously' full. If the next data packet may cause an overflow,

the main processor is suspended until more space becomes available. It should be clear that all such optimizations depend on an accurate model of the actual hardware.

7 THE WATCHDOG TASK SYSTEM ON SINGLE PROCESSOR ARCHITECTURES

At first glance, scheduling both task MAIN and the watchdog task on the same processor seems contradictory to the rationale behind the use of a watchdog. There is, however, a common application where this is valuable.

As mentioned in Sec. 1.1, the timing in real-time systems is critical, so it is usually not possible to insert assertions into the code. Such systems, however, characteristically have idle time. The watchdog task can be scheduled during these idle intervals and can execute the assertions previously inserted by the main task into the queue. In this scheme, the assertions become analogous to the background checking routinely performed in real time systems.

The generality of the Watchdog Task system is demonstrated by the fact that such an arrangement requires no changes to the software. To install the system on a single processor, it is sufficient to assign a low priority to the watchdog and a high priority to the main task by using the PRIORITY pragma. This results in the correct scheduling, at least on the Data General MV/10000 machine where the system was implemented.

The queue that is used for communication on the single processor is not restricted since it resides in a large main memory accessed by two processes which are mutually exclusive. In this special case, the

linked list of packets implementation of the queue (Fig. 6.7b), which was too costly for multiprocessor architectures, becomes feasible.

The latency problem is even more difficult to control in the single processor architecture. In this arrangement, task MAIN may not be able to wait for the watchdog to catch up before a critical action. The situation may be partially relieved if the idle time intervals are frequent enough that the watchdog task is never too far behind, and if the critical actions are relatively sparse. These 'optimistic circumstances' are probably not unrealistic in typical real time systems. An even further improvement may be possible by tagging critical assertions and executing only these when there is limited time available.

8 ANOTHER APPLICATION FOR THE WATCHDOG TASK SYSTEM

Consider the situation depicted in Fig. 8.1. Task MAIN in system_1 calls a procedure that has been transformed into system_2. The execution would proceed smoothly: The MAIN task of system_2 runs sequentially with MAIN_1. The watchdog of system_1 is idle, and it can be suspended while the watchdog of system_2 is scheduled. Once system_2 has finished executing, watchdog_1 will be resumed. The ability to 'nest' systems provides a further dimension of flexibility.

This suggests a further application for the Watchdog Task system: Libraries of Ada packages that have assertions built into them can be preprocessed and stored as Watchdog Task systems, ready to be called by other Watchdog Task systems. This would result in a very convenient environment for reliable software systems employing

watchdogs.

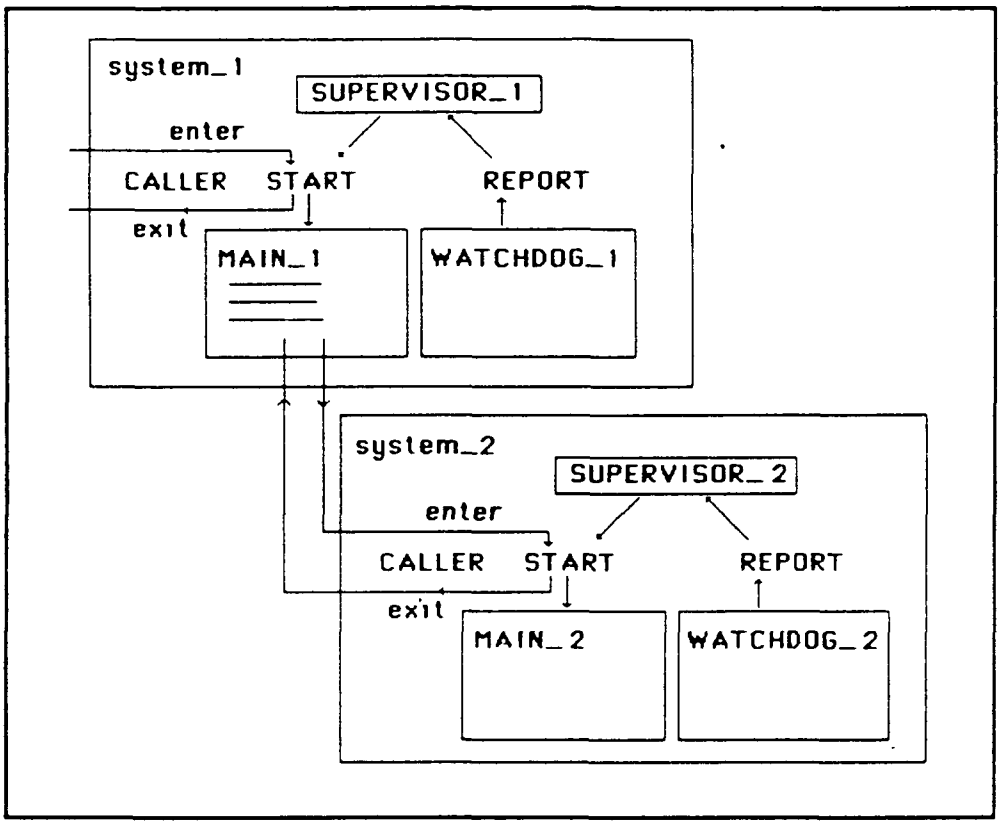

Fig. 8.1 Nesting Watchdog Task Systems

8

9 SPECIAL PURPOSE WATCHDOG TASK SYSTEMS

[Mahmood 85] categorizes watchdog processors into two categories: Special purpose and general purpose. The same distinction can be made of Watchdog Task systems. The system that has been presented in this paper falls into the general purpose category; it is meant to be applicable to all sequential programs and to run on all common architectures. There may be applications where it is possible to specialize the Watchdog Task system, resulting in a reduction of cost and an improvement in efficiency.

Special features of the application, for example, may simplify communication. Consider a situation where the main program changes global variables only during well defined intervals. A watchdog system can be designed for this application that has no communication overhead. Task MAIN no longer needs to insert data in a queue; the watchdog simply accesses the globals during the 'stable' intervals. This example demonstrates the extent of the improvement possible in special cases.

10 THE PREPROCESSOR

The preprocessing that transforms the sequential main program to the Watchdog Task system is straightforward. Once the assertions have been replaced by queue insertions and the watchdog task has been built, all that remains is to arrange everything into the predefined structure. More careful preprocessing, however, can reduce the runtime overhead significantly.

Assertions quite often occur in groups in the main program. Some of these assertions may be referencing the same program variables, the values of which stay constant from assertion to assertion (since assertions do not have any side effects). It is clearly profitable to merge the data transfer for such assertions together. This will not only reduce the total amount of data that gets transferred, but will also decrease the per assertion overhead of requesting space from the queue, since this will be done for a group of assertions.

During preprocessing, it is also possible to detect variables that do not get changed between certain assertions. The value of such

variables can be stored in the watchdog and used for later assertions, further reducing the data transfer. Having the watchdog store state may have other benefits: 'Virtual variables' (which can be used, for example, to store the values of program variables) are usually inserted into the main program to measure changes in variables between different instantiations of the same assertion. Anna [Luckham 84] provides support for extensive use of virtual Ada code. At least some of the overhead of the virtual code can be taken over by the watchdog.

In summary, the more work put into preprocessing, the more significant the reduction in runtime overhead becomes.

SUMMARY AND CONCLUSIONS

The Watchdog Task is a software tool analogous to the Watchdog processor. The Assertion Watchdog processor can be implemented as a multitasking software system in a high level language. The main motivation behind a software implementation is the flexibility obtained by this high level approach. The software system can also be used to model different hardware architectures.

The underlying structure of a Watchdog Task system in ADA was presented. The system provides a recovery mechanism and a tool for controlling latency. It was shown that the efficiency of the system depended critically on communication. The communication problem, which is meaningful only in multiprocessing, is solved by mapping the architectural communication (usually just a buffer) into software. Variations and optimizations were discussed.

The Watchdog Task System also runs on single processor architectures as a type of background checking mechanism. This is an attractive new application for the watchdog, but it has certain difficulties, particularly a problem of latency, associated with it.

The issues of preprocessing, special requirements on the compiler and scheduler, and an application for the watchdog system in reliable libraries of packages were also discussed in this paper.

ACKNOWLEDGEMENTS

Special thanks are extended to Aamer Mahmood for his guidance throughout this research. The authors are grateful to Doug Bryan, Prof. D. H. Luckham and the Anna group for their invaluable help with Ada. Acknowledgements are also due to Saied Bozourgi-Nesbat for careful review of this report and to Lydia Christopher, Samiha Mourad and the Rats group for their support.

This work was partially supported by NASA-AMES under Contract No. NAG 2-246 and by the National Science Foundation under Grant No. DCR-8200129.

REFERENCES

[Andrews 81] Andrews, D. M., and J. P. Benson, "An Automated Program Testing Methodology and its Implementation," Proceedings of the 5th International Conference on Software Engineering, pp. 254-261, San Diego, California, March 9-12, 1981.

[Daniels 83] Daniels, S. F., "A Concurrent Test Technique for Standard Microprocessors," Digest of Papers, Compcon Spring 83, pp. 389-394, San Francisco, California, February 28-March 3, 1983.

[Eifert 84] Eifert, J. B., and J. P. Shen, "Processor Monitoring Using Asynchronous Signatured Instruction Streams," Digest, 14th International Conference on Fault Tolerant Computing (FTCS-14), pp. 394-399, Kissimmee, Florida, June 20-22, 1984.

[[Lu 80] Lu, D. J., "Watchdog Processors and VLSI", Proceedings of the National Electronics Conference, Vol. 34, pp. 240-245, Chicago, Illinois, October 27-28, 1980.

[Lu 82] Lu, D. J., "Watchdog Processor and Structural Integrity Checking," IEEE Transactions on Computers, Vol. C-31, No. 7, pp. 681-685, July 1982.

[Luckham 84] Luckham D., and F. W. Henke, "An Overview for Anna-A Specification Language for Ada," Computer Systems Laboratory Technical Report 84-265, Stanford, California, September 1984.

[Mahmood 83] Mahmood, A., D. J. Lu, and E. J. McCluskey, "Concurrent Fault Detection using a Watchdog Processor and Assertions," Proceedings 1983 International Test Conference, pp. 622-628, Philedelphia, Pennsylvania, October 18-20, 1983.

[Mahmood 84] Mahmood, A., D. M. Andrews, and E. J. McCluskey, "Executable Assertions and Flight Software," Proceedings of the AIAA/IEEE 6th Digital Avionics Systems Conference, pp. 346-351, Baltimore, Maryland, December 3-6, 1984.

[Mahmood 85] Mahmood, A., and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors - A Survey," CRC Technical Report No. 85-7, Stanford, California, May 1985.

[Namjoo 82a] Namjoo, M., and E. J. McCluskey, "Watchdog Processors and Capability Checking," Digest of Papers, 12th Annual International Symposium on Fault Tolerant Computing (FTCS-12), pp. 245-248, Santa Monica, California, June 22-24, 1982.

[Namjoo 82b] Namjoo, M., "Techniques for Concurrent Testing of VLSI Processor Operation," Digest, 1982 International Test Conference, pp. 461-468, Philedelphia, Pennsylvania, November 15-18, 1982.

8

[Namjoo 83] Namjoo, M.,"Cerberus-16: An Architecture for a General Purpose Watchdog Processor," Digest of Papers, 13th Annual International Symposium on Fault Tolerant Computing (FTCS-13), pp. 216-219, Milano, Italy, June 28-30, 1983.

[Randell 75] Randell, B., System Structure for Software Fault Tolerance," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pp. 220-232, June 1975.

[Saib 79] Saib, S. H., "Distributed Architectures for Reliability," Proceedings of the AIAA Computers in Aerospace Conference, pp. 458-462, Los Angeles, California, October 22-24, 1979.

[Shen 83] Shen, J. P., and M. A. Schuette, "On-Line Self-Monitoring Using Signatured Instruction Streams," Proceedings 1983 International Test Conference, pp. 275-282, Philedelphia, Pennsylvania, October 18-20, 1983.

[Sridhar 82] Sridhar, T., and S. M. Thatte, "Concurrent Checking of Program Flow in VLSI Processors," Digest, 1982 International Test Conference, pp. 191-199, Philedelphia, Pennsylvania, November 15-18, 1982.

APPENDIX

```
-- The SUPERVISOR package

   with TEXT_IO;
   package SUPERVISOR is

       NUMBER_OF_ASSERTIONS : constant := 4;
       type ERROR_RANGE is range 1 .. NUMBER_OF_ASSERTIONS;
       procedure START;
       procedure REPORT (ERROR : ERROR_RANGE);
       function IS_DONE return BOOLEAN;

       pragma INLINE(IS_DONE);

   end SUPERVISOR;


   package body SUPERVISOR is

       type ACTION is range 0 .. 2;
       CURRENT_ACTION : ACTION;
       DONE : BOOLEAN;


       task MAIN_PROGRAM is
          --pragma PRIORITY(10); The pragma is necessary for proper
                                -- scheduling on a single processor
          entry START;
       end MAIN_PROGRAM;

       task body MAIN_PROGRAM is separate;

       function IS_DONE return BOOLEAN is

       begin
          RETURN DONE;
       end IS_DONE;

       procedure UPDATE_CURRENT_ACTION is

       begin
          CURRENT_ACTION := CURRENT_ACTION + 1;
       end UPDATE_CURRENT_ACTION;
```

```
        procedure START is

        begin

         CURRENT_ACTION := 0;
         loop
             begin
                 case CURRENT_ACTION is
                     when 0      =>  MAIN_PROGRAM.START;
                     when others =>  TEXT_IO.PUT ("no recovery specified");
                 end case;
                 EXIT;
             exception
                 when TASKING_ERROR =>
                     UPDATE_CURRENT_ACTION;
                     TEXT_IO.PUT ("error caught by WT");
                     TEXT_IO.NEW_LINE;
                 when others =>
                     UPDATE_CURRENT_ACTION;
                     TEXT_IO PUT ("exception propagated from MP"),
                     TEXT_IO.NEW_LINE;
             end;
         end loop;
         DONE := TRUE;
     end START;

     procedure REPORT (ERROR : ERROR_RANGE) is

     begin
         case ERROR is
             when 1 =>  TEXT_IO.PUT ("error 1");
             when 2 =>  TEXT_IO.PUT ("error 2");
             when 3 =>  TEXT_IO.PUT ("error 3");
             when 4 =>  TEXT_IO.PUT ("error 4");
         end case;
         TEXT_IO.NEW_LINE;
         abort MAIN_PROGRAM;
     end REPORT;

begin
     DONE := FALSE;
end SUPERVISOR;
```

```
-- The format of the Main task


with WATCHDOG;
separate (SUPERVISOR)

    task body MAIN_PROGRAM is

        WAIT_TIME : constant := 1.0;

        -- internal declarations are inserted here


        procedure WAIT_FOR_WATCHDOG is

           begin
               while not WATCHDOG.IS_IDLE loop
                       delay WAIT_TIME;
               end loop;
           end WAIT_FOR_WATCHDOG;

        pragma INLINE(WAIT_FOR_WATCHDOG);

    begin
        accept START do

            -- body of the main program

            WAIT_FOR_WATCHDOG;
        end START;
    end MAIN_PROGRAM;
```

-- Example call to the Watchdog with the unit-data queue implementation

---

```
ASSERT_1:
begin
  WATCHDOG.QUEUE_REQUEST_SPACE(FOR_NEXT => 3);
  WATCHDOG.QUEUE.INSERT(LABEL => 1);
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(HDG_ERROR));
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(TAS_MS));
  WATCHDOG.QUEUE.INSERT(FL_DATA => FLOAT(LAT_LIM_CMD));
end ASSERT_1;
```

-- Example call to the Watchdog with the packet queue implementation

---

```
ASSERT_1:
declare
    PACKET : WATCHDOG.QUEUE.DATA_ARRAY (1 .. 3);
begin
    PACKET (1) := (KIND    => WATCHDOG.QUEUE.FL_POINT,
                   FL_DATA => FLOAT (HDG_ERROR));
    PACKET (2) := (KIND    => WATCHDOG.QUEUE.FL_POINT,
                   FL_DATA => FLOAT (TAS_MS));
    PACKET (3) := (KIND    => WATCHDOG.QUEUE.FL_POINT,
                   FL_DATA => FLOAT (LAT_LIM_CMD));
    WATCHDOG.QUEUE.REQUEST_SPACE;
    WATCHDOG.QUEUE INSERT
       (LABEL => 1, NUMBER_OF_PARS => 3, DATA => PACKET);
end ASSERT_1;
```

```
-- Format of the Watchdog with the packet queue implementation

with SUPERVISOR;
package WATCHDOG is

    function IS_IDLE return boolean;

    package QUEUE is
       type DATA_TYPE is (INT, FL_POINT);
       type UNIT_DATA (KIND : DATA_TYPE := FL_POINT) is
           record
               case KIND is
                   when INT =>
                      INT_DATA : INTEGER;
                   when FL_POINT =>
                      FL_DATA : FLOAT;
               end case;
           end record;

       MAX_NUMBER_OF_PARS : constant := 10;
       type PAR_RANGE is range 1 .. MAX_NUMBER_OF_PARS;

       type DATA_ARRAY is array (PAR_RANGE range <>) of UNIT_DATA;

       MAX_LABEL : constant := 4;
       type LABEL_TYPE is range 1 .. MAX_LABEL;

       function  IS_EMPTY return BOOLEAN;
       procedure REQUEST_SPACE;
       procedure INSERT   (LABEL          : LABEL_TYPE;
                           NUMBER_OF_PARS : PAR_RANGE;
                           DATA           : DATA_ARRAY);
       procedure GET_NEXT (LABEL : out LABEL_TYPE);
       procedure GET_NEXT (DATA : out DATA_ARRAY);
       procedure FLUSH;

       pragma INLINE (INSERT,GET_NEXT,IS_EMPTY,FLUSH,REQUEST_SPACE);

    end QUEUE;


end WATCHDOG;
```

```
package body WATCHDOG is

    package body QUEUE is separate;

function IS_IDLE return BOOLEAN is
    begin
      RETURN QUEUE.IS_EMPTY;
    end IS_IDLE;


task CHECK is
    -- pragma PRIORITY(0); The pragma is necessary for proper
                          -- scheduling on a single processor
end CHECK;

task body CHECK is
    WAIT_TIME : constant := 1.0;
    LABEL : QUEUE.LABEL_TYPE;

begin
    loop
            if QUEUE.IS_EMPTY then
                 if SUPERVISOR.IS_DONE then EXIT;
                  else delay WAIT_TIME;
                 end if;
            end if;
    while not QUEUE.IS_EMPTY loop
            QUEUE.GET_NEXT (LABEL);
            case LABEL is
                when 1 =>
                   ASSERT_1:
                   declare
                       PACKET : QUEUE.DATA_ARRAY (1 .. 3);
                   begin
                       QUEUE.GET_NEXT (PACKET);
                       if (abs (PACKET (1).FL_DATA *
                                PACKET (2).FL_DATA) >= 0.02442) then
                          if abs (abs (PACKET (3).FL_DATA) - 0.5) >
                             0.0001 then
                              SUPERVISOR.REPORT (ERROR => 1);
                              QUEUE.FLUSH;
                          end if;
                       end if;
                   end ASSERT_1;
```

```
                when 2 =>
                    ASSERT_2:
                        -- evaluate the second assertion
                    end ASSERT_2;

                     -- evaluate the rest of the assertions

                end case;
            end loop;
        end loop;
    end CHECK;

end WATCHDOG;
```

```
-- Format of the Watchdog with the unit-data queue implementation

with SUPERVISOR;
package WATCHDOG is

   function IS_IDLE return BOOLEAN;

   package QUEUE is
      type DATA_TYPE is (INT, FL_POINT);
      type UNIT_DATA (DATA_KIND : DATA_TYPE := FL_POINT) is
          record
              case DATA_KIND is
                  when INT =>
                     I_DATA : INTEGER;
                  when FL_POINT =>
                     F_DATA : FLOAT,
              end case;
          end record;

      MAX_NUMBER_OF_PARS : constant := 10;
      type PAR_RANGE is range 1 .. MAX_NUMBER_OF_PARS;

      type DATA_ARRAY is array (PAR_RANGE range <>) of UNIT_DATA;

      MAX_LABEL : constant := 4;
      type LABEL_TYPE is range 1 .. MAX_LABEL;

      function  IS_EMPTY return BOOLEAN;
      procedure REQUEST_SPACE (FOR_NEXT : PAR_RANGE);
      procedure INSERT   (LABEL          : in LABEL_TYPE);
      procedure INSERT   (FL_DATA        : in FLOAT);
      procedure INSERT   (INT_DATA       : in INTEGER);
      procedure GET_NEXT (LABEL : out LABEL_TYPE);
      procedure GET_NEXT (NO_OF_PARS : in PAR_RANGE; DATA : out DATA_ARRA
      procedure FLUSH;

      pragma INLINE (INSERT,GET_NEXT,IS_EMPTY,IS_FULL,FLUSH,REQUEST_SPACE)

   end QUEUE;


end WATCHDOG;
```

```
package body WATCHDOG is

   package body QUEUE is separate;

function IS_IDLE return BOOLEAN is
   begin
     RETURN QUEUE.IS_EMPTY;
   end IS_IDLE;

task CHECK is
   -- Pragma PRIORITY(0); The pragma is necessary for proper
                        -- Scheduling on a single processor
end CHECK,

task body CHECK is
   WAIT_TIME : constant := 1.0;
   LABEL : QUEUE.LABEL_TYPE;

begin
   loop
       if QUEUE.IS_EMPTY then
          if SUPERVISOR.IS_DONE then EXIT;
          else delay WAIT_TIME,
           end if;
       end if;
       while not QUEUE.IS_EMPTY loop
          QUEUE.GET_NEXT (LABEL);
          case LABEL is
              when 1 =>
                 ASSERT_1:
                 declare
                     PACKET : QUEUE.DATA_ARRAY (1 .. 3);
                 begin
                     QUEUE.GET_NEXT (NO_OF_PARS => 3, DATA => PACKET)
                     if (abs (PACKET (1).F_DATA *
                              PACKET (2).F_DATA) >= 0.02442) then
                        if abs (abs (PACKET (3).F_DATA) - 0.5) >
                           0.0001 then
                            SUPERVISOR.REPORT (ERROR => 1);
                            QUEUE.FLUSH;
                        end if;
                     end if;
                 end ASSERT_1;
              when 2 =>
                 ASSERT_2:
                     -- evaluate the second assertion
                 end ASSERT_2;
              -- evaluate the rest of the assertions
```

```
                end case;
            end loop;
        end loop;
    end CHECK;

end WATCHDOG;
```

```
separate (WATCHDOG)
-------------------------------------------------------------------

-- Queue implementation : Linked list of packets

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

package body QUEUE is

    type PACKET (NUMBER_OF_PARS : PAR_RANGE);
    type PACKET_POINTER is access PACKET;
    type PACKET (NUMBER_OF_PARS : PAR_RANGE) is
       record
           L    : LABEL_TYPE;
           D    : DATA_ARRAY (1 .. NUMBER_OF_PARS);
           NEXT : PACKET_POINTER;
       end record;

    type LINKED_LIST is
       record
           FRONT : PACKET_POINTER;
           REAR  : PACKET_POINTER;
       end record;

    LIST : LINKED_LIST;

    function IS_EMPTY return BOOLEAN is

    begin
       RETURN (LIST.FRONT.NEXT = null);
    end IS_EMPTY;


    procedure REQUEST_SPACE is

    begin
       null; -- Assuming the linked list never fills up
    end REQUEST_SPACE;

    procedure INSERT (LABEL          : LABEL_TYPE;
                      NUMBER_OF_PARS : PAR_RANGE;
                      DATA           : DATA_ARRAY) is

       TEMP : PACKET_POINTER;
```

```
    begin
        TEMP := new PACKET (NUMBER_OF_PARS);
        TEMP.L := LABEL;
        TEMP.D := DATA;
        TEMP.NEXT := null;
        LIST.REAR.NEXT := TEMP;
        LIST.REAR := TEMP;
    end INSERT;

    procedure GET_NEXT (LABEL : out LABEL_TYPE) is

    begin
        LABEL := LIST.FRONT.NEXT.L;
    end GET_NEXT;

    procedure GET_NEXT (DATA : out DATA_ARRAY) is

    begin
        DATA := LIST.FRONT.NEXT.D;
        LIST.FRONT := LIST.FRONT.NEXT;
    end GET_NEXT;

    procedure FLUSH is

    begin
        LIST.FRONT.NEXT := null;
        LIST.REAR := LIST.FRONT;
    end FLUSH;

begin
    LIST.FRONT := new PACKET (1);
    FLUSH;
end QUEUE;
```

```
separate (WATCHDOG)
-----------------------------------------------------------------

-- Queue implementation . Ring buffer of packets

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

package body QUEUE is

    BUFFER_SIZE   constant  = 100,
    WAIT_TIME     constant  = 1 0,

    type PACKET (NO_OF_PARS : PAR_RANGE .= 1) is
       record
           L · LABEL_TYPE,
           D   DATA_ARRAY (1 .. NO_OF_PARS);
       end record;

    type PACKET_ARRAY is array (INTEGER range 0 .  BUFFER_SIZE - 1) of PAC

    type RING_BUFFER is
       record
           ITEMS : PACKET_ARRAY;
           FRONT : INTEGER range 0 .. BUFFER_SIZE - 1;
           REAR  : INTEGER range 0 .. BUFFER_SIZE - 1;
       end record;

   BUFFER : RING_BUFFER,

   procedure REQUEST_SPACE is

      function BUFFER_IS_FULL return BOOLEAN is

      begin
          RETURN ((BUFFER.REAR + 1) mod BUFFER_SIZE = BUFFER.FRONT);
      end BUFFER_IS_FULL;

      pragma INLINE (BUFFER_IS_FULL),

   begin
      while BUFFER_IS_FULL loop
          delay WAIT_TIME;
      end loop;
   end REQUEST_SPACE,
```

```
    function IS_EMPTY return BOOLEAN is

    begin
       RETURN (BUFFER.REAR = BUFFER.FRONT);
    end IS_EMPTY;


    procedure INSERT (LABEL          : LABEL_TYPE;
                      NUMBER_OF_PARS : PAR_RANGE;
                      DATA           : DATA_ARRAY) is

    begin
       BUFFER.ITEMS (BUFFER.REAR) :=
         (NO_OF_PARS => NUMBER_OF_PARS, L => LABEL, D => DATA);
       BUFFER.REAR  = (BUFFER.REAR + 1) mod BUFFER_SIZE;
    end INSERT;

    procedure GET_NEXT (LABEL : out LABEL_TYPE) is

    begin
       LABEL := BUFFER.ITEMS (BUFFER FRONT).L;
    end GET_NEXT;

    procedure GET_NEXT (DATA : out DATA_ARRAY) is

    begin
       DATA := BUFFER.ITEMS (BUFFER.FRONT).D;
       BUFFER.FRONT := (BUFFER FRONT + 1) mod BUFFER_SIZE;
    end GET_NEXT;

    procedure FLUSH is

    begin
       BUFFER.FRONT := BUFFER_SIZE - 1;
       BUFFER.REAR := BUFFER_SIZE - 1;
    end FLUSH;

begin
    FLUSH;
end QUEUE;
```

```
separate (WATCHDOG)
-------------------------------------------------------------

-- Queue implementation - Ring buffer of unit-data

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

package body QUEUE is


    BUFFER_SIZE   constant := 20;
    WAIT_TIME     constant := 10;

    type ITEM_TYPE is (LABL, DATA);
    type ITEM (KIND : ITEM_TYPE := DATA) is
       record
           case KIND is
              when LABL =>
                 L : LABEL_TYPE;
              when DATA =>
                 D : UNIT_DATA;
           end case;
       end record;

    type ITEM_ARRAY is array (INTEGER range 0 .. BUFFER_SIZE - 1) of ITEM;

    type RING_BUFFER is
       record
           ITEMS : ITEM_ARRAY;
           FRONT : INTEGER range 0 .. BUFFER_SIZE - 1;
           REAR  : INTEGER range 0 .. BUFFER_SIZE - 1;
       end record;

    BUFFER : RING_BUFFER;

    function IS_EMPTY return BOOLEAN is

    begin
       RETURN (BUFFER.FRONT = BUFFER.REAR);
    end IS_EMPTY;

    procedure REQUEST_SPACE (FOR_NEXT : PAR_RANGE) is

       function BUFFER_IS_FULL (FOR_NEXT_PARS : PAR_RANGE) return BOOLEAN
```

```
        TEMP    INTEGER;
    begin
        -- an item in the buffer is sacrificed to simplify
        -- overflow checking

        TEMP := INTEGER (BUFFER.REAR) + 1 + INTEGER (FOR_NEXT_PARS);
        if BUFFER.REAR < BUFFER.FRONT then
            RETURN (TEMP >= BUFFER.FRONT);
        else
            RETURN ((TEMP > BUFFER_SIZE - 1) and
                    (TEMP mod BUFFER_SIZE >= BUFFER.FRONT));
        end if;
    end BUFFER_IS_FULL;

    pragma INLINE (BUFFER_IS_FULL);

begin
    while BUFFER_IS_FULL (FOR_NEXT) loop
        delay WAIT_TIME;
    end loop;
end REQUEST_SPACE;

procedure INSERT (LABEL : LABEL_TYPE) is

begin
    BUFFER.ITEMS (BUFFER.REAR) := (KIND => LABL, L => LABEL);
    BUFFER.REAR := (BUFFER.REAR + 1) mod BUFFER_SIZE;
end INSERT;

procedure INSERT (FL_DATA : FLOAT) is

begin
    BUFFER.ITEMS (BUFFER.REAR) :=
      (KIND => DATA, D => (DATA_KIND => FL_POINT, F_DATA => FL_DATA));
    BUFFER.REAR := (BUFFER.REAR + 1) mod BUFFER_SIZE;
end INSERT;

procedure INSERT (INT_DATA : INTEGER) is

begin
    BUFFER.ITEMS (BUFFER.REAR) :=
      (KIND => DATA, D => (DATA_KIND => INT, I_DATA => INT_DATA));
    BUFFER.REAR := (BUFFER.REAR + 1) mod BUFFER_SIZE;
end INSERT;
```

```
    procedure GET_NEXT (LABEL : out LABEL_TYPE) is

    begin
       LABEL := BUFFER.ITEMS (BUFFER.FRONT).L;
       BUFFER.FRONT := (BUFFER.FRONT + 1) mod BUFFER_SIZE;
    end GET_NEXT;

    procedure GET_NEXT (NO_OF_PARS : PAR_RANGE; DATA : out DATA_ARRAY) is

    begin
       for I in 1 .. NO_OF_PARS loop
           DATA (I) := BUFFER.ITEMS (BUFFER.FRONT).D;
           BUFFER.FRONT := (BUFFER.FRONT + 1) mod BUFFER_SIZE;
       end loop;
    end GET_NEXT;

    procedure FLUSH is

    begin
       BUFFER.FRONT := BUFFER_SIZE - 1;
       BUFFER.REAR := BUFFER_SIZE - 1;
    end FLUSH,

begin
    FLUSH;
end QUEUE;
```

```
separate (WATCHDOG)
-------------------------------------------------------------------

-- Queue implementation : The double buffer

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

package body QUEUE is


    BUFFER_SIZE . constant := 50;

    type ITEM_TYPE is (LABL, DATA);
    type ITEM (KIND : ITEM_TYPE := DATA) is
       record
           case KIND is
              when LABL =>
                  L : LABEL_TYPE;
              when DATA =>
                  D : UNIT_DATA;
           end case;
       end record;

    type ITEM_ARRAY is array (INTEGER range 0 .. BUFFER_SIZE - 1) of ITEM;
    -- Simple arrays are sufficient for the double buffer
    -- implementation

    type BUFFER is
       record
           ITEMS : ITEM_ARRAY;
           FRONT : INTEGER range 0 .. BUFFER_SIZE;
           REAR  : INTEGER range 0 .. BUFFER_SIZE;
       end record;

    BUFFER : array (BOOLEAN) of RING_BUFFER;

    OF_MP  : BOOLEAN;

    function OF_WT return BOOLEAN is

    begin
       RETURN (not OF_MP);
    end OF_WT;
```

```
    pragma INLINE (OF_WT);

    function IS_EMPTY return BOOLEAN is

    begin
       RETURN (BUFFER (OF_WT).FRONT = BUFFER (OF_WT).REAR);
    end IS_EMPTY;

    procedure REQUEST_SPACE (FOR_NEXT : PAR_RANGE) is

       function MP_BUFFER_IS_FULL (FOR_NEXT_PARS : PAR_RANGE) return BOOL

          TEMP : INTEGER;
       begin
          TEMP := INTEGER (BUFFER (OF_MP).REAR) + 1 + INTEGER (FOR_NEXT_PA
          RETURN (TEMP > BUFFER_SIZE);
       end MP_BUFFER_IS_FULL;

       procedure SWAP is

       begin
          OF_MP := not OF_MP;
          BUFFER (OF_MP).FRONT := 0;
          BUFFER (OF_MP).REAR := 0;
       end SWAP;

       pragma INLINE (MP_BUFFER_IS_FULL, SWAP);

    begin
-- The swapping of the buffers is performed by the
-- Main Task (producer) because this eliminates the
-- need for locking the queue. Task Main swaps the
-- buffers only when the Watchdog buffer is empty.

       if MP_BUFFER_IS_FULL (FOR_NEXT) then
          while not IS_EMPTY loop
             null;
          end loop;
          SWAP;
       end if;
    end REQUEST_SPACE;

    procedure INSERT (LABEL : LABEL_TYPE) is

    begin
```

8

```
    BUFFER (OF_MP).ITEMS (BUFFER (OF_MP).REAR) :=
      (KIND => LABL, L => LABEL);
    BUFFER (OF_MP).REAR := (BUFFER (OF_MP).REAR + 1) mod BUFFER_SIZE;
end INSERT;

procedure INSERT (FL_DATA : FLOAT) is

begin
    BUFFER (OF_MP).ITEMS (BUFFER (OF_MP).REAR) :=
      (KIND => DATA, D => (DATA_KIND => FL_POINT, F_DATA => FL_DATA));
    BUFFER (OF_MP).REAR := BUFFER (OF_MP).REAR + 1;
end INSERT;

procedure INSERT (INT_DATA : INTEGER) is

begin
    BUFFER (OF_MP).ITEMS (BUFFER (OF_MP).REAR) :=
      (KIND => DATA, D => (DATA_KIND => INT, I_DATA => INT_DATA));
    BUFFER (OF_MP).REAR := BUFFER (OF_MP) REAR + 1;
end INSERT;

procedure GET_NEXT (LABEL : out LABEL_TYPE) is

begin
    LABEL := BUFFER (OF_WT).ITEMS (BUFFER (OF_WT).FRONT).L;
    BUFFER (OF_WT).FRONT := BUFFER (OF_WT).FRONT + 1;
end GET_NEXT;

procedure GET_NEXT (NO_OF_PARS : PAR_RANGE; DATA : out DATA_ARRAY) is

begin
    for I in 1 .. NO_OF_PARS loop
        DATA (I) := BUFFER (OF_WT).ITEMS (BUFFER (OF_WT).FRONT).D;
        BUFFER (OF_WT).FRONT := BUFFER (OF_WT).FRONT + 1;
    end loop;
end GET_NEXT;

procedure FLUSH is

begin
    BUFFER (OF_WT).FRONT := 0;
    BUFFER (OF_WT).REAR := 0;
    BUFFER (OF_MP).FRONT := 0;
    BUFFER (OF_MP).REAR := 0;
end FLUSH;
```

```
begin
    FLUSH;
    OF_MP := TRUE;
end QUEUE;
```