Implementing Robot Controllers under Real-Time POSIX and Ada

By: M. González Harbour, J.M. Drake Moyano, M. Aldea Rivas, and J. García Fernández
Departamento de Electrónica y Computadores
Universidad de Cantabria
39005- Santander, SPAIN
{mgh,drakej}@ctr.unican.es, {mario,jgf}@ctrhp6.ctr.unican.es

Abstract[1]

In this paper we present our experience in the development of real-time controllers for special robots, designed to perform maintenance operations in nuclear power plants. The evolution of computer hardware and software technology has made the requirements of the embedded robot controllers to also evolve. In addition to the usual concurrent software and real-time constraints, today our robots require advanced features such as network communications, file systems, and graphical user interfaces. To meet all these requirements we not only need a reliable concurrent programming language such as Ada, but we also need to use services provided by real-time operating systems. In this paper we want to present our successful experience in the development of robot controllers using the Ada language and a POSIX real-time operating system.

Keywords: *POSIX, Ada, Real-Time Applications, Robot Controllers, Real-Time Operating Systems*

1. Introduction

At the Computers and Real-Time Systems Group at the University of Cantabria, we have been involved in the development of robot software for several years. During these years we have evolved through different generations of development platforms and tools that represent different stages of software technology.

Our first controllers (1985-1992) were built using a real-time cross Pascal compiler which used HP 9000/300 workstations as a host, and VME boards based on 680x0 microprocessors as the target platform. The software architecture was based on a cyclic executive model, which made the application difficult to implement and maintain.

Between 1990 and 1995 we used a cross Ada compiler with the same host and target platform; it allowed us to write our code using the Ada tasking concurrency model. The timing behaviour of this kind of fixed-priority concurrent software can be predicted by using *Rate Monotonic Analysis* [1]. Although the Ada run-time system introduced a significant degree of overhead for the 16 bit microprocessors that we were using, our software was still able to meet its timing requirements, and was much more reliable and easier to maintain thanks to the use of Ada. However, we found ourselves writing software for basic services, specially communications software. Besides, there was an increasing demand to include in the robot controllers advanced features that were outside the scope of the Ada language, such as graphical user interfaces, or a file system (which was not available on the embedded platform). Moreover, in the larger applications we had software modules written in different programming languages and that were developed by different companies. For these modules it was found convenient to provide some level of protection to disable them from accidentally accessing each others internal data. All these services could be provided by the use of a real-time operating system conforming to the POSIX standard. At the time we decided to use a real-time OS the relevant POSIX standards [3][4] were not approved, but there were POSIX-like OS implementations already in the market [5][6].

The experience that we have had with this kind of development platform has been very successful. We have continued using the Ada programming language as much as possible, and we have used the services provided by the operating system. This allows us to produce software that meets its real-time requirements and that is reliable and easy to maintain, while providing all the advanced features that are common today in non real-time systems.

In this paper we want to present this experience, and we want to focus on the specific mechanisms that we have used

1. This work has been funded by the *Comisión Interministerial de Ciencia y Tecnología* of the Spanish Government under grant TAP94-0996, and by *Equipos Nucleares S.A.*

of to achieve our goals. In Section 2 we will make a quick review of the functional and timing requirements of our robot controllers. In Section 3 we will explain the main issues that we have faced in the use of the Ada language, in particular regarding task synchronization. In Section 4 we will discuss the main operating system-related issues, in particular the inter-program synchronization, and the program scheduling issues. In Section 5 we will discuss the special characteristics of the HP-RT platform, which we are using in our current projects. Finally, in Section 6 we will give a brief overview of the robot controllers that we have designed using Ada and real-time operating systems.

2. Functional and Timing Requirements

Most of our robots perform maintenance operations in nuclear power plants. This implies that the control system is divided in two units: a *local controller*, which is located next to the robot and inside the radioactive area, and the *teleoperation station*, where the operator supervises and/or controls the robot from a non-radioactive area. Both units are typically 100 meters apart, and communicate by wire or fiber optic cable.

The typical functional requirements for the teleoperation station include a graphical user interface (GUI), interfaces to special-purpose keyboards and joysticks, sensor data filtering and processing, image processing, and communications with the local controller. For the local controller, the main functions are the coordinated control of several joints, sensor reading and filtering, digital and analog output controls, and communications to the teleoperation station and also to a local terminal and other local devices. Figure 1 shows the basic functional and timing requirements for both units.

Many of the tasks that execute both in the teleoperation station and in the local controller have real time requirements of different kinds. The local controller is an embedded computer and it has the strictest timing requirements. It has hard real-time requirements for the servo motor control algorithm (5 ms), trajectory planning

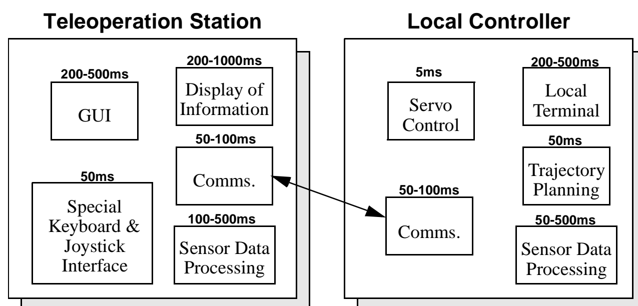

Figure 1. Functional and timing requirements

(50 ms), reaction to user commands (100 ms), and reception and transmission of control messages (50-100 ms). It also has soft real-time requirements for those tasks related to the processing of display information for the local terminal (200-500 ms), and the transmission of data for display and post-processing (50-500 ms).

The teleoperation station also has hard and soft real-time requirements. The hard requirements are related to the reading of joysticks and a special-purpose keyboard (50 ms), and the transmission of control messages (50-100 ms). The soft real-time requirements are related to the display of information (200-1000 ms), and the reception of sensor data and status information (100-500 ms). It also has to perform non-real time activities, such as processing and storage of sensor data, and report generation.

3. Ada Language Issues

If we look at the commercial tools available nowadays for developing real-time systems, we find that there are two major choices for implementing a real-time system based on a concurrent model. We can use the C programming language plus real-time operating system services to implement concurrency (such as POSIX threads), or we can use Ada. We’ve had experience with both approaches, and for us the C/POSIX threads model is very error-prone. Ada tasking is part of the language, and thus the compiler makes a lot of checks, and helps in building correct and more reliable concurrent software.

However, the Ada 83 language had some well-known drawbacks for real-time applications. The ones that affect our applications are data synchronization being inefficient and with unbounded priority inversion, scheduling being too inflexible, and interrupt handling being inefficient. Fortunately, most real-time Ada vendors currently supply non-standard extensions that help in solving these problems. The Ada-95 language also solves these problems: data synchronization and interrupt handling is now more efficient through protected objects; unbounded priority inversion can be avoided by using priority ceiling locking for protected objects; and the scheduling model has the required flexibility. In addition, Ada 95 incorporates new technology that we think that will be very useful to real-time systems, such as the OOP capabilities, and hierarchical libraries. We will be able to use all these new features in a portable way when real-time Ada 95 compilers become commercially available for the platforms that we use; but, for now, we have to continue using the Ada-83 compilers that are available today.

The synchronization mechanism provided in Ada 83 —the rendezvous— is inefficient for data synchronization and has

priority inversion. Most real-time Ada compilers provide a more efficient synchronization mechanism by defining an optimization pragma (usually called *Monitor* or *Passive*) that allows a task with certain restrictions to be handled as a synchronization object, with no thread of control of its own, and similar to the Ada 95 protected objects. However, although this allows a much more efficient synchronization in terms of the average case response time, in many of these implementations the priority model of the task is not implemented, and thus severe priority inversions may arise. One solution to that problem is to implement at the application level the priority ceiling algorithm that is used in the Ada 95 protected objects. For this purpose it is necessary that the compiler provides the capability of dynamically modifying the priority of a thread, which is an extension with respect to the Ada 83 standard that is usually provided by real-time compilers.

In order to keep the synchronization mechanism as safe as possible, we have implemented the synchronization objects through a monitor that wraps all the synchronization and priority change operations in a uniform and safe way. One problem that we found with this solution was that if an exception was raised inside the monitor task, that task would terminate causing the application to fail. A global exception handler could not be put in place because the task did not have a thread of control of its own. Therefore, we had to catch the exceptions that occurred in each `accept` statement with an internal exception handler, and convert the exception value to a parameter that could be passed to the caller. The resulting pseudocode can be seen in Table 1. It can be seen that the treatment of exceptions is clumsy, but fortunately the Ada 95 protected objects will bring the right solution with respect to the handling of priorities and of the exceptions. The rules of the Ada language protect this package against the abortion of the calling task by deferring abortion until the end of the accept statement, once it has started. If we wish to have several protected objects with the same operations and attributes, we could implement the package shown as a generic package, with the `Ceiling_Priority` as a generic formal parameter; in this case, each object would be an instantiation of the generic package.

Table 1. Pseudocode of a data synchronization object

```
package Protected_Object is
   procedure P1(...);
   procedure P2(...);
end Protected_Object;

package body Protected_Object is

   Ceiling_Prio:constant System.Priority:=...;
   type Error_Cause is
      (No_Error,Cause1,Cause2,...);

   task Object_Monitor is
      entry E1(...,Error : out Error_Cause);
      entry E2(...,Error : out Error_Cause);
      pragma Monitor;
   end Object_Monitor;

   procedure P1(...) is
      Old_Prio : System.Priority:=Get_Priority;
   begin
      Set_Priority(Ceiling_Prio);
      Object_Monitor.E1(...,Error);
      Set_Priority(Old_Prio);
      case Error is
         when No_Error => null;
         when Cause1 => raise Exception1;
         ...
      end case;
   end P1;

   procedure P2(...) is ...;
      -- identical structure

   task body Object_Monitor is
   begin
      loop
         select
            accept E1
               (...,Error:out Error_Cause)
            do begin
               -- do work;
               Error:=No_Error;
            exception
               when Exception1=> Error:=Cause1;
               ...
            end E1;
         or
            accept E2 ...;
         or
            terminate;
         end select;
      end loop;
   end Object_Monitor;

end Protected_Object;
```

4. Need for a Real-Time Operating System

Since we are using Ada that is a concurrent language which supports the concurrency, synchronization, scheduling and timing requirements of real-time applications, why do we need a real-time OS? The answer is that there are features that are outside the scope of the language and that are required for our applications. We need to use operating system services, such as a file system, network communications, and graphical user interfaces. In addition, in the larger applications that we have built, we found it very convenient to have memory protection between the different parts of the application, especially for those parts that were written in different (and less reliable) languages (C), and for those parts developed by different companies. Memory protection can be accomplished if the operating system supports concurrent execution of multiple programs

or partitions, and supports the real-time scheduling and synchronization that is required to have the ability of meeting all the timing requirements.

Since portability is a very important issue for us given the ever-changing nature of hardware, a good choice is a real-time POSIX OS that supports both POSIX.1b (the real-time extensions to POSIX [2]), and POSIX.1c (the threads extension [3]). The threads extension allows a large part of the Ada run-time system to be built on top of operating system services. If Ada tasks are built on top of OS threads the scheduling of tasks of different processes (programs) can be consistent across the entire system, which is extremely useful in multi-program applications. This is the approach taken in the Thomson compiler for the HP-RT operating system [8] that we have used for our robot controllers.

Caution is necessary when selecting a POSIX conforming operating system for real-time systems development. It is necessary to know that POSIX.1b or POSIX.1c conformance are not synonymous of real-time behaviour. The POSIX standard only specifies the interface to the operating system services and their semantics, but does not specify how they are implemented. This means that the system services may or may not be implemented in a way that has predictable timing behaviour. Therefore, we need to get from the vendor, from our own benchmarks, or from the user community, the bounds on the response time for each of the services that will be used in the real-time part of our software.

We must also have in mind that most of the services described in POSIX.1b and POSIX.1c are optional, and that the standard leaves many behaviours as "unspecified", or "implementation defined". We must always check that the services that we need are present, and that the behaviours that we need are supported. In the following subsections we will focus on particular issues that are very important to be checked for real-time applications.

4.1. Scheduling issues

The POSIX standard gives the implementation a large amount of freedom to chose how the different processes and threads will be scheduled. Basically there are two mechanisms that are defined in POSIX for scheduling threads:

- A thread may have a *process-wide contention scope*, which means that it only contends for the CPU with threads within it's same process. The process itself is scheduled at a different level, contending with other processes.
- A thread may have a *system-wide contention scope*, which means that it contends for the CPUs with all the other threads that have system-wide contention scope, independently of the process to which they belong

An implementation must support process-wide or system-wide contention scopes, but it is not required to support both. But for multi-program and multi-threaded (or multi-task) real-time applications, process-wide is very difficult and unsafe to use, because the priorities of the different threads or tasks is not consistent across different processes. For example, it could happen that the highest priority active task could be preempted by a lower priority task belonging to a different process, just because the OS chose to schedule the latter process at that time. Therefore, if we want to accomplish predictable scheduling in a multi-program application, we must make sure that the system-wide contention scope is supported.

The control systems that we use for robots are not multiprocessor systems, but if they were so we would have to pay attention to the *allocation domain* of each thread, which is defined as the set of processors on which that thread may execute. The POSIX scheduling rules are only predictable if the allocation domain of threads is static, i.e., each thread is statically allocated to only one processor. The way to specify the allocation domain is implementation defined, and we must make sure that the implementation allows a static allocation domain for threads.

4.2. Synchronization Issues

When programming a real-time application in Ada we can use the Ada-language mechanisms for task synchronization within a program or partition, but we cannot use it to synchronize with tasks belonging to different programs or partitions, in which case we must use the OS synchronization services.[1]

POSIX.1b and POSIX.1c specify several mechanisms that can be used to achieve both mutual exclusive synchronization (counting semaphores and mutexes) and signal & wait synchronization (counting semaphores, and condition variables). For real-time applications it is necessary to know that if we use counting semaphores to achieve mutual exclusion, since they do not have any support for priority inheritance or priority ceiling protocols, very severe priority inversions might happen, thus making

1. It may be possible to use the active or passive partitions described in the Ada 95 Distributed Systems Annex to synchronize tasks belonging to different partitions, but this is problematic for portable real-time applications since the scheduling policies, the treatment of priorities, and the management of shared resources between partitions are implementation defined.

our application unable to meet its timing requirements. Mutexes however do have optional support for priority inheritance and/or priority ceiling (which is called *priority protection* in POSIX). Another behaviour that is optional for mutexes is the ability to use them for threads belonging to different processes (process-shared mutexes).

Therefore we must make sure that the process-shared option and one of the priority inheritance or priority protect options are supported by our implementation. If not, it is still possible to use the counting semaphores for mutual exclusion, by using an application-level mechanism to increase the priority of the thread locking the semaphore (before the lock occurs), and returning to the previous priority after unlocking the semaphore, in a similar way as that indicated by Table 1.

Apart from the synchronization mechanism, in order to share data between different processes, which by default have independent address spaces, it is necessary to create a shared memory object in which the shared data is allocated. Table 2 shows the pseudocode of a package that provides for a shared data object accessible from more than one process. We can see that three operations are needed to access the shared memory object: first it must be opened and created if necessary; then we must set its size using the `Truncate_File` POSIX call; finally, the shared memory object must be mapped into the process's address space. The mapping call returns the address of the object, which is then used in a representation clause to create the corresponding Ada variable.

4.3. Timing Behaviour Issues

The first aspects that we have to check against the requirements of our real-time application are the resolutions of the operating system clocks, the Ada `Calendar.Clock` function, and the `delay` sentence (which are usually the same). The POSIX standard only requires 20 ms for the clock resolution (although the implementation may support resolutions down to 1 ns), which is not usually enough for most real-time systems. If the clock resolution is smaller than the deadlines or the timing requirements of our application, there is very little that can be done except to add an extra hardware clock to the system, or change to a different platform.

Even if the clock has enough resolution, we need to check that the required timing requirements are met. It is well known that the Ada 83 relative `delay` sentence is not adequate for implementing periodic activations of task, which is a very common requirement in real-time systems (Ada 95 introduces the new absolute `delay until` construction that avoids this problem). One solution to this problem is to use the POSIX timers, which can be programmed to measure a relative interval or an absolute time, or can also be programmed to expire periodically. Table 3 shows the pseudocode of a package that can be used to create a periodic timer and to activate a periodic task based upon it. A user of this package must remember that the handling of signals and threads is a little clumsy in

Table 2. Pseudocode of a process-shared synchronization object

```
function Create_Object (Name : String;
         Length : Integer)
         return System.Address is
   Fd : POSIX_IO.File_Descriptor;
   Start_Addr : System.Address;
begin
   Fd:=POSIX_Shared_Memory_Objects.
      Open_Or_Create_Shared_Memory
         (Name => To_POSIX_Str(Name),...);
   POSIX_IO.Truncate_File
      (Fd, POSIX.IO_Count (Length));
   Start_Addr:=POSIX_Memory_Mapping.Map_Memory
      (Storage_Offset (Length), Protection,
       POSIX_Memory_Mapping.Map_Shared, Fd, 0);
   return Start_Addr;
end Create_Object;

package Process_Shared_Object is
   procedure P1(...);
   procedure P2(...);
end Process_Shared_Object;

package body Process_Shared_Object is

   Ceiling_Prio:constant System.Priority:=...;
   Semaphore :
      POSIX_Semaphores.Semaphore_Descriptor;
   Object_Name : String :="/shared_object";
   type Object_Type is ...;
   Length : Integer:=Object_Type'Size;
   Object_Address : constant System.Address:=
      Create_Object(Object_Name,Length);
   Shared_Object : Object_Type;
   for Shared_Object use at Object_Address;

   procedure P1(...) is
      Old_Prio : System.Priority:=Get_Priority;
   begin
      Set_Priority(Ceiling_Prio);
      POSIX_Semaphores.Wait(Semaphore);
      --do work with shared_object
      POSIX_Semaphores.Post(Semaphore);
      Set_Priority(Old_Prio);
   end P1;

   procedure P2(...) is ...;
      -- identical structure

begin

   Semaphore:=POSIX_Semaphores.Open_Or_Create(
      Name=>To_POSIX_Str(Object_Name&".sem"),
      Value=>1,...);

end Process_Shared_Object;
```

Table 3. Periodic activation operations

```
generic
   Signal_Used : POSIX_Signals.Signal;
package Periodic_Activation is
   procedure Initiate (Period:Duration);
   procedure Wait_For_Next_Period;
   Already_Initiated : exception;
end Periodic_Activation;

package body Periodic_Activation is

   The_Id : POSIX_Timers.Timer_Id;
   The_Set : POSIX_Signals.Signal_Set;
   Initiated : Boolean:=False;

   procedure Initiate (Period:Duration) is
      Event : POSIX_Signals.Signal_Event;
      State : POSIX_Timers.Timer_State;
   begin
      if Initiated then
         raise Already_Initiated;
      else
         Initiated:=True;
         POSIX_Signals.Delete_All_Signals(
            The_Set);
         POSIX_Signals.Add_Signal(
            The_Set,Signal_Used);
         POSIX_Signals.Set_Notification(
            Event,
            POSIX_Signals.Signal_Notification);
         POSIX_Signals.Set_Signal
            (Event,Signal_Used);
         The_Id:=POSIX_Timers.Create_Timer(
            POSIX_Timers.Clock_Realtime,Event);
         POSIX_Timers.Set_Initial(
            State,To_POSIX_Timespec(Period));
         POSIX_Timers.Set_Interval(
            State,To_POSIX_Timespec(Period));
         POSIX_Timers.Arm_Timer (
            Timer => The_Id,
            Options=>
               Timer_Options(POSIX.Empty_Set),
            New_State => State);
      end if;
   end Initiate;

   procedure Wait_For_Next_Period is
      Sig : POSIX_Signals.Signal;
   begin
      Sig:=POSIX_Signals.Await_Signal(The_Set);
   end Wait_For_Next_Period;

end Periodic_Activation;

task Periodic_Task is
   package The_Timer is new Periodic_Activation
      (POSIX_Signals.Realtime_Signal'FIRST);
      -- for example
begin
   The_Timer.Initiate(Period);
   loop
      -- do useful work
      The_Timer.Wait_For_Next_Period;
   end loop;
end Periodic_Task;
```

POSIX, because all tasks in the process must mask or block the signal used by the timer. If any individual thread forgets to mask it or inadvertently unmasks it at a later stage, the signal generated by the timer will be delivered to the wrong thread.

4.4. Systems Programming Issues

Many real-time systems use special I/O hardware and therefore it is usually necessary to be able to write your own device drivers. The POSIX standard does not specify the way in which device drivers can be programmed or used, and thus this is left as an implementation dependent part of the application. It is important that the operating system provides the user with enough information to allow him to write his own device drivers, including the handling of hardware interrupts. The non portability of device drivers may soon be solved because a standardization effort called Uniform Driver Interface (UDI) is currently under way for device drivers [7]. This industry standard will provide platform- and operating system-independent interfaces for device driver implementation. It is being developed by many of the main computer software and hardware companies.

In addition, since standard device drivers such as disk I/O, ethernet communications, terminal I/O, etc., are going to be used in a real-time application together with the application-specific drivers, it is extremely important that the system provides a way to choose the priority at which each of these device drivers run. Otherwise, it would be very difficult to guarantee the timing response of tasks with tight deadlines. An ethernet driver, for example, may sometimes take several milliseconds to execute and, if this execution is performed at a high priority level, this may compromise the response of a task with deadlines in the millisecond range.

5. The HP-RT platform

The HP-RT platform is based on an HP-PA 7100 processor on an VME bus board, and running under a POSIX-like real-time operating system (a port of the LYNX operating system [5] to the HP processor). It has a cross development system in which the host computer is a conventional HP workstation. The OS is POSIX-like because it was developed before the real-time POSIX standards were approved, and thus it follows old drafts of these standards. It is expected that now that the standards are approved, a fully conforming version will appear. The OS comes with Xwindows software, sockets for LAN communications, POSIX threads, and all the real-time POSIX services.

With respect to the options that have been mentioned in this paper as being important for real-time applications, we can briefly state the level of support for each of them:

- *Scheduling*: All threads are scheduled with system-wide contention scope.
- *Synchronization*: Process-shared mutexes are not supported, and thus we need to use semaphores for inter-process synchronization. Although the implementation supports priority inheritance for semaphores, this is not a standard feature and, besides, the worst-case response time for the priority ceiling synchronization is better. For this reason we have implemented inter-process synchronization according to the pseudocode given in Table 2.
- *Timing*: The OS provides a low resolution "system clock" (10 ms) and three high resolution clocks (1 µs). All of them may be used to create timers.
- *I/O Drivers*: They may be programmed by the application developer, may handle hardware interrupts, and may use their own threads of control, with a user-assigned priority. System-provided drivers can also be assigned the desired priorities.
- *Embeddable*: The file system may reside in RAM, and the system may boot from a LAN or from non-volatile memory (through a PCMCIA interface)

The Ada compiler that we have used is the Thomson cross compiler for HP-RT, which has the following characteristics:

- Each Ada task is associated with an OS thread. This introduces some inefficiency, but it allows a global scheduling of all the tasks, independently of the program to which they belong. This allows a real-time multi-process application to meet its timing requirements, while providing memory protection between the different processes.
- The compiler provides the pragma `Monitor` optimization for data synchronization, but it does not support priority ceiling and, therefore, a scheme like the one shown in Table 1 must be used.
- The `delay` statement is bound to the low resolution clock (10 ms), and there is no support for `delay until`, so a scheme using POSIX timers like the one shown in Table 3 should be used.

6. Results Obtained

Using the HP-RT and the LYNX OS platforms we have developed three robot controllers for Equipos Nucleares S.A, funded in part by the Spanish Government:

- *Mobile Teleoperated Robot* (RMT). Joint project with Equipos Nucleares S.A. and Construcciones Aeronáuticas S.A. The robot is designed for maintenance operations inside nuclear power plants. It is a mobile robot, with a six-joint arm, and video and radiation sensing capabilities. The teleoperation station has two units, one for the control of the robot and display of information, and the other one for video image processing. The embedded local controller was originally programmed in C, but an Ada implementation is now being developed.
- *Telemanipulated Arm*: It is a six-joint arm designed for inspection and maintenance operations in nuclear power stations. The embedded controller acts both as the teleoperation station and the robot controller. The commands from the operator are input through a remote non-intelligent special-purpose command unit.
- *CASEVA*: It is a five-joint arm designed for welding pieces for nuclear power stations. The pieces are very thick and hundreds or thousands of welding layers are needed. The trajectory of the welding tool must be controlled depending on the location of the previous welding layers, and thus image processing is required to control the trajectory.

The results obtained have been extremely successful. A single robot controller architecture was designed, and this has allowed us to reuse a large portion of the controller for building the others. The HP-RT operating system together with the Thomson compiler allow us to meet all the timing requirements of the application. The HP-PA processor is extremely fast and has allowed us a quick development of the trajectory planning software, using floating point arithmetic. In previous controllers where floating point arithmetic was very expensive, the use of integer arithmetic made this part of the application extremely complex and non portable.

7. Conclusion

Recent advances in software technology have enabled the implementation of portable and reliable real-time software. The concurrent approach is much more flexible and maintainable than the cyclic executive model. Fixed priority scheduling is currently supported both in the Ada and POSIX standards, and Rate Monotonic Analysis allows us to predict the timing behaviour of fixed-priority concurrent

software. The use of Ada with is support both for concurrency and software engineering principles has played an essential role in the success of our implementations. The use of a standard real-time OS facilitates advanced features such as GUIs, network communications, etc.

The HP-RT platform, although is not yet strictly conforming to the POSIX standards, presents operating systems services that have a predictable timing response, together with other services of modern operating systems, such as network communications, file system, Xwindows, etc. The high resolution timing services provided allow sub-millisecond timing requirements to be met. The software developed is quite portable, although it will be much more portable when the platform evolves to Ada 95 and the final POSIX standards. I/O drivers would remain non portable for the moment, until a standard is approved in this area, and the corresponding implementations become available.

References

[1] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. González Harbour. "*A Practitioner's Handbook for Real-Time Analysis*". Kluwer Academic Pub., 1993.

[2] IEEE Standard 1003.1b:1993, *"Information Technology - Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) -- Amendment 1: Realtime Extension [C Language]*". Institute of Electrical and Electronic Engineers, 1993.

[3] ISO/IEC Standard 9945-1:1996 (E), *"Information Technology -Portable Operating System Interface (POSIX)- Part 1: System Application Program Interface (API) [C Language]*". Institute of Electrical and Electronic Engineers, 1996.

[4] IEEE Std. 1003.5b:1996, "*Information Technology -POSIX Ada Language Interfaces- Part 1: Binding for System Application Program Interface (API)-- Amendment 1: Realtime.* Institute of Electrical and Electronic Engineers, 1996

[5] Lynx Real-Time Systems Inc. "LynxOS User's Manual", 1992.

[6] Hewlett Packard. "HP-RT Reference Manual", 1994

[7] Project UDI Working Group. "Uniform Driver Interface. Environment Specification", 1996.

[8] Thomson Software Products. "Adaworld for HP Series 700 Workstations under HP-UX to HP-PA under HP-RT: Application Developer's Guide", 1993.