The GNARL Implementation of POSIX/Ada Signal Services (extended abstract)

Dong-Ik Oh T.P. Baker Seung-Jin Moon

Department of Computer Science
Florida State University
Tallahassee, FL 32306-4019
Phone: (904)644-3441
Internet: doh@cs.fsu.edu

Abstract

This paper describes the POSIX signal model, the standard C and Ada language signal interfaces, and implementation of application-level signal handling in the GNu Ada Runtime Library (GNARL). GNARL maps some signals to exceptions, and allows others to be handled via protected procedures, task entries, or the POSIX.5b synchronous signal-awaiting operations. *In the final paper, performance data will also be provided, to guide users in choosing between these various interfaces.*

1 Introduction

The GNu Ada Runtime Library (GNARL) is the tasking runtime library of the Gnu NYU Ada'95 [1] Translator (GNAT)[8]. For portability, GNARL is divided into two layers, the lower of which isolates dependences on a particular host operating system or real-time kernel. The primary implementation of the GNARL lower level accesses OS services via the POSIX.1[1, 5, 6] interfaces.

POSIX.1 is a family of standard C-language application program interfaces to operating system services, developed by the IEEE in cooperation with ISO/IEC JTC1/SC22/WG15. POSIX.5 is the Ada language binding for POSIX.1. The POSIX API is derived from that of UNIX, and is at least partially supported by the many operating systems derived from UNIX. New OS releases continue to converge toward the POSIX interface, as most major vendors have committed to eventually conform to it.

Ada applications that execute in the environment of a POSIX or other UNIX-like operating system must deal with signals, the software interrupts that the OS may deliver to an application at virtually any time. Thus, it is essential that the Ada runtime system provide some form of application-level signal handling mechanism.

[1] We use the term Ada'95 for the Ada language standard which was adopted by ISO/IEC in 1995[3] and Ada'83 for the 1983 ANSI standard, which was endorsed by ISO/IEC in 1987[7].

This paper describes the four signal handling mechanisms supported by GNARL, and how they are implemented. It also provides performance figures that may guide application developers when they choose which of these mechanisms to use.

The remainder of this paper is organized as follows. Section 2 explains the POSIX signal model and its variants, from both the C and the Ada viewpoints, and the relationship of signals to Ada interrupts. Section 3 explains the signal handling mechanisms supported by GNARL, and how they are implemented. Section 4 points out the inherent limitations of this design, and presents some performance results. Section 5 concludes, with a summary of the GNARL implementation status and plans for future development.

2 Background

This section provides some background on the semantic model of POSIX signals, and the differences in the ways signals are seen by programs written in Ada versus C.

2.1 POSIX Signals

A POSIX signal is a form of software interrupt. Much of the semantics of signals is independent of the programming language, but the primary standards describe the semantics from the viewpoint of a C program.

There is a finite number of signals, which are identified to C programs by integers. The exact set of signals is dependent on the OS implementation. POSIX defines names for certain signals and specifies certain conditions under which they are generated. It also specifies certain subprograms which a C-language application can call to control the generation and delivery of signals.

Signals may be used for notification of a variety of events:

1. **Run-time error.** committed by a program. For example, a signal may be generated by a division by zero, a floating-point overflow, a memory protection violation, a reference to a non-existent memory location, or an attempt to execute an illegal instruction. (These signals need to be mapped to exceptions by the Ada runtime system. See section 2.2.)
2. **Time-out.** Operations are provided to request that a signal be generated when a clock reaches a specified time, or when a specified span of time has elapsed.
3. **I/O completion or failure.** Asynchronous input and output operations generate a signal when an operation completes, or if an operation fails.
4. **Job control.** A user or process may suspend, resume, and terminate the execution of a process by sending it certain signals. A user may do this by hitting certain keys on the terminal that is controlling the process.
5. **Interprocess communication.** A process may send a signal to another process to notify it of an event.
6. **Interthread communication.** In a system that supports the POSIX Threads option, a thread may send a signal to another thread within the same process to notify it of an event.

Much of the specific semantics of signals can vary from one POSIX-compliant operating system to another. Part of this variation is due to explicit implementation options which were introduced by successive revisions of the POSIX.1 standard. POSIX.1[1] defines a basic signal model. This is extended by POSIX.1b[5] to include new operations and a new kind of "real time" signal, and then modified to account for multi-threaded processes by POSIX.1c[6]. Thus, the exact signal model supported by a given POSIX-compliant operating system may vary, depending on whether it supports the Realtime Signals and/or Threads options.

In the POSIX.1 and POSIX.1b models, the kind of entity to which a signal may be delivered is a process. In the POSIX.1c model the kind of entity to which a signal may be delivered is a thread of control (within a process); a distinction is made between operations that generate a signal for a process (so that it can be delivered to any thread within the process) and other operations that generate a signal for a specific thread (so that it can only be delivered to that one thread).

Certain signals can be masked. In POSIX.1 and POSIX.1b, when a signal is generated for a process and the process has the signal masked, the signal remains pending until the process unmasks it. A process manipulates its signal mask by calls to the C-language function *sigprocmask()*. POSIX.1c changes this for systems that support the Threads option, by making the signal mask part of the state of each individual thread, rather than the entire process; the effect of *sigprocmask()* becomes implementation-dependent, and the new function *pthread_sigmask()* is introduced to manipulate the calling threads signal mask.

Ordinarily only one pending instance of a masked signal is required to be retained; that is, if a signal is generated N times while it is masked the number of signal instances that are delivered to the process when it finally unmasks the signal may be any number between 1 and N. POSIX.1b introduces a special kind of "realtime signal', if the Realtime Signals option is supported, that guarantees instances generated by certain operations will always be retained.

Each signal is associated with some signal action. The binding of signal action to signal is part of the state of each process. POSIX defines the default action for each signal, which may be to ignore the signal, terminate the process, stop the process, or continue the process. (POSIX does not specify whether a signal that is generated is ignored or pended, if the signal action is ignored and the signal is also masked.) For some signals the default action cannot be overridden. For other signals, the application may specify a handler procedure that is called, asynchronously, when the signal is delivered to the process. Such a handler interrupts the execution of the process or thread at the instant the signal is delivered. The process can specify the desired action for a signal via the C-language *sigaction()* call; it can specify a handler procedure, restore the default action, or specify that the signal should be ignored.

POSIX.1b introduces the *sigwaitinfo()* call, for systems where the Realtime Signals option is supported, and POSIX.1c introduces the *sigwait()* call, for systems where the Threads option is supported. These C-language function calls both allow a process or thread to *accept* any one of a specified set of masked signals. They come in both blocking and non-blocking forms. The use of this new synchronous delivery interface is especially recommended for multithreaded processes, because the thread synchronization operations defined by POSIX.1c are not safe for use within an asynchronous signal handler.

POSIX.1 defines the *kill()* function that generates an instance of a signal for a specified process or group of processes. POSIX.1b adds the *sigqueue()* function, that generates a lossless instance of a signal for a processor group of processes, if the Realtime Signals option is supported. POSIX.1c adds *pthread_kill()*, which generates an instance of a signal to a specified thread, if the Threads option is supported.

2.2 Signals and Ada

POSIX.5[2] is the standard Ada'83 language binding for POSIX.1. Because POSIX.1 processes execute in disjoint virtual address spaces, POSIX.5 specifies that all the tasks of a program are part of a single POSIX process. Because POSIX.1 does not recognize multiple threads of control within a process, POSIX.5 defines the effect of POSIX calls by a process on Ada tasks within the process, including the effects on signal delivery.

POSIX.5 defines the basics of an Ada'83 application's view of POSIX signals. Certain signals are reserved for use by the Ada runtime system; some of these are to be mapped to Ada exceptions, and some are reserved for use in implementing delay statements. `Block_Signals` and `Unblock_Signals` operations are provided for masking and unmasking signals, and `Ignore_Signal` and `Unignore_Signal` operations are provided for setting the signal action to ignore a signal. Because of Ada scoping and concurrency complications, an Ada application cannot directly bind a subprogram as asynchronous handler for a signal, but it may use address clauses to bind signals to task entries. The Ada runtime system is expected to implement this using its own low-level asynchronous handler subprograms.

The Ada'83 interrupt entry mechanism associates an interrupt with a task entry via an address clause. The binding is established during the elaboration of the task and detached during the finalization of the task. For example, a POSIX signal SIGQUIT can be associated with a task entry as follows:

```
Addr : constant System.Address
  := Ada.Interrupts.Reference
    (Ada.Interrupts.Names.SIGQUIT);

task Handler is
  entry Done;
  for Done use at Addr;
end Handler;
```

A signal that is bound to a task entry is effectively masked except at times when the handler task is at an accept statement for the signal entry. If the task executes an accept while the signal is pending, or the signal arrives while the task is waiting at the accept, the effect is that of a call to the entry; that is, the accept body is executed and the handler task continues with its execution.

POSIX.5b is the draft revision of POSIX.5 to reflect POSIX.1b and POSIX.1c. POSIX.1c defines the behavior of multiple threads of control within a process, if the Threads option is supported. If the OS supports the Threads option it is generally a good thing for Ada applications, since if Ada tasks are implemented as POSIX threads blocking system calls will not block other tasks in the program. However, the behavior that POSIX.1c specifies for C threads conflicts in some respects with the behavior that POSIX.5 and the Ada Reference Manual[3] specify for Ada tasks. POSIX.5b deals with these differences by hiding most of them, and making a few changes to the POSIX.5 signal model.

In the case of signals, POSIX.5b is forced to modify the semantic model of POSIX.5, to allow for per-task signal masking. However, it cannot *require* per-task signal masking, since support for Threads is an OS-dependent option. Therefore, the effect of `Block_Signals` and `Unblock_Signals` is allowed to be either per-task or per-process. Likewise, it cannot require per-thread signal delivery, so there is no Ada operation corresponding directly to *pthread_kill()*. The closest thing is the `Interrupt_Task` operation, which may be implemented using *pthread_kill()* but may also be implemented in the Ada runtime system if the Threads options is not supported by the OS.

In Ada'95, interrupt entries are classified as obsolescent, and their use is "not recommended". The recommended way of handling interrupts is via protected procedures. An interrupt can be associated with a parameterless protected procedure in two ways. It can be associated statically via the pragma `Interrupt_Handler` or can be associated dynamically through a procedure call `Attach_Handler` provided in the language defined Ada.Interrupts package. The static binding is similar to that of interrupt entries; the binding takes effect during the elaboration of the protected object which contains the procedure and its finalization. Dynamic binding is effective from the point where a protected procedure is being bounded to an interrupt up until a corresponding unbinding operation is made for the same interrupt ID through services provided in the package Ada.Interrupts.

If signals are viewed as interrupts, it would seem logical for an Ada'95 POSIX binding to provide a mechanism for attaching protected procedures to signals, so that upon delivery of a signal that is attached to a protected procedure the protected procedure will be called. An example of the static and dynamic binding mechanisms applied to POSIX signals is given below.

```
with Ada.Interrupts.Names;
package Protected_Unit is
  protected Handlers is
    procedure Handler1;
    procedure Handler2;
    pragma Attach_Handler
      (Handler1, Ada.Interrupts.Names.SIGQUIT);
    -- static binding during the elaboration
    pragma Interrupt_Handler (Handler2);
    -- Handler2 can be attached to a
    -- signal dynamically
  end Handlers;
end Protected_Unit;

with Ada.Interrupts;
with Ada.Interrupts.Names;
with Protected_Unit;
procedure Main is
  H : Ada.Interrupts.Parameterless_Handler
    := Handlers.Handler2'access;
begin
  ...
  Attach_Handler (H, Ada.Interrupts.Names.SIGUSR2);
  -- Handler2 is dynamically attached
  -- to the POSIX SIGUSR2 signal.
  ...
  Detach_Handler (H);
  ...
end Main;
```

While this interface can be implemented (and is implemented by GNAT/GNARL), POSIX.5b does not take this approach. One reason is that, with POSIX.1c, the semantics of signals have changed, so that they no longer fit the Ada interrupt model. The Ada model assumes that the interrupt associated with a protected procedure can be masked during all protected operations of the associated protected object, and that this or some other mechanism can be used to prevent the handler from interrupting the other protected operations of the protected object. In a POSIX.1c multithreaded process, this is not possible. There is no operation that simultaneously masks a signal for all threads, and there is no locking operation that can be safely called from an asynchronous signal handler. Thus, the required Ada semantics cannot be implemented by having a POSIX signal handler procedure directly call a protected procedure. The other reason that POSIX.5b does not use the Ada'95 interrupt handler mechanism for signals is that amendments to POSIX currently in ballot would add interfaces for true hardware interrupt handlers, which behave differently from signals. It was felt that using the same interface for both hardware interrupts and signals would be confusing.

The preferred application-level signal handling interface in POSIX.5b is a binding to *sigwait()*, the synchronous delivery mechanism. An Ada task calls `Await_Signal` or `Await_Signal_With_Timeout` to wait for a pending or arriving instance in a specified set of signals. The effect is similar to an Ada accept or selective wait statement, but the procedure-call syntax does not allow mixing in accepts for Ada task entries or a terminate alternative, so the implementation can be a direct mapping to *sigwait()* if that operation is supported by the OS. POSIX.5b also retains the POSIX.5 signal entry interface.

3 GNARL Implementation

In this section we explain how GNARL implements POSIX signal services. All the mechanisms mentioned above are supported:

1. **Exceptions.** Certain *reserved* signals are mapped to standard Ada exceptions. SIGSEGV is mapped to `Storage_Error` or `Constraint_Error`, depending on context. SIGFPE and SIGILL are mapped to `Constraint_Error`. SIGABRT is mapped to a special [2] exception which is used to implement both whole-task abortion and the implementation of asynchronous transfer of control (the asynchronous select statement).

2. **Await signal.** The GNARL uses *sigwait()* internally, to implement other operations, as explained below. Applications are also able to access this functionality directly using `Await_Signal`. `Await_Signal` is provided by the FSU implementation of POSIX.5b, which makes use of GNARL internals but is not included in the standard GNAT distribution.

3. **Signal handlers.** A protected procedure may be attached to a signal, using the interfaces shown in

[2]The special feature is that this exception cannot be handled by ordinary application code.

the example in Section 2.2. A special thread of control, created implicitly by the Ada runtime system, "accepts" the signal using *sigwait()*, and then calls the associated protected entry.

4. **Signal entries.** A task entry may be attached to a signal, using an address clause as specified in POSIX.5 and POSIX.5b. As with signal handlers, a special thread of control accepts the signal and calls the associated task entry.

The mapping of the reserved signals to Ada exceptions is implemented by asynchronous signal handler procedures that are attached using *sigaction()* by the runtime system (RTS), and which propagate the appropriate exception. In the case of SIGABRT, the signal handler only propagates the special exception `Abort_Signal` if abort is not deferred for that task. Otherwise, a flag is set that is checked at every point abort is undeferred; if the flag is set, the undeferral of abort raises the exception. Further discussion of the GNAT implementation of exceptions is outside the scope of this paper. See [4] for more details.

For signal handlers (and signal entries), it may seem wasteful to interpose a separate thread of control. Specifically, one can imagine using *sigaction()* to install a low-level asynchronous handler procedure that directly calls the protected procedure (or executes the accept body). The signal would be kept unmasked in one server task and masked in all the rest of the tasks. The server task might be the environment task (or the task that owns the signal entry). This approach does not work, for several reasons. The main problem is implementing mutual exclusion, to prevent concurrent execution of the handler with other operations of the protected object (or the acceptance of other entry calls by the task).

As we mentioned above, the primary implementation of the GNARL lower-level is implemented using POSIX threads. In this situation, we have per-thread signal masking, per-process signal actions, and do not have any thread-level locking primitives that we can call from inside an asynchronous signal handler.

In a POSIX multithreaded environment, we cannot rely on masking the signal for mutual exclusion, because there is no way for a task who executes another protected action of the protected object to mask the signal in the server task. On the other hand, the signal handler procedure cannot call the POSIX thread-level locking primitives to lock the protected object (or lock the task control block of the task that owns the signal entry), because those operations are not async-signal safe[3]. Therefore, the only safe way to "handle" a signal is via the *sigwait()* operation in a server thread. The server thread wakes up when the signal arrives, and calls the associated protected procedure or task entry.

There is a choice is between dedicating one server task for all signals and providing a server task for each signal. The former approach looks attractive, since it saves runtime space, but it will block other signals during the rendezvous or protected entry call. This may result in delayed or lost signals. For this reason, GNARL provides a separate server task for each application-level signal handler. A simple control structure for such a task might be:

```
loop
  pthread_sigmask (SIG_BLOCK, signal_set, old_set);
  sig := sigwait (signal_set, Signal);
  if (Handler_Installed (Signal)) then
    <Make a call to a handler>
    -- Make an entry or protected procedure call
    -- depending on the type of handler installed
  end if;
end loop;
```

This produces behavior that is acceptable when there is a handler installed, but we need to allow the handler to be attached and detached dynamically. If the signal is received when there is no handler attached we want to take the default action. We cannot achieve this effect so long as the handler task is sitting on the *sigwait()*. Even if we have used *sigaction()* to set the asynchronous signal action to the default, that action will not be taken unless the signal is unmasked, and we cannot unmask the signal while the server is blocked on *sigwait()* because in POSIX.1c the effect of *sigwait()* is undefined if any of the signals for which it called is unmasked. Therefore, when the application-level handler is detached from the signal, we must wake up the handler task and cause it to wait instead on some operation for which it is safe to leave the signal unmasked, so that the default action can be taken. One possible implementation might be as follows:

```
loop
  if (not Handler_Installed (Signal)) then
    sigaction (Signal, default_action, old_action);
    pthread_sigmask (SIG_UNBLOCK, signal_set,
old_set);
    pthread_mutex_lock (mutex);
    pthread_cond_wait (cond, mutex);
```

[3]If the mutex locking operation were async-signal safe, there would still be a potential problem with deadlock when the signal interrupts a call to another protected operation by the same task. In principle, this might be prevented using the per-task signal mask, since it is the same task.

```
    -- Wait to take a default action until a
    -- handler needs to be installed
    pthread_mutex_unlock (mutex);
    pthread_sigmask (SIG_BLOCK, signal_set,
old_set);
  else sig := sigwait (signal_set, Signal);
    if (Handler_Installed (Signal)) then
      <Make a call to an entry or a handler>
      -- Make an entry or protected procedure call
      -- depending on the type of handler installed
    end if;
end loop;
```

In this way a handler task can correctly reflect the signal handling status change. The attaching and detaching operations for a signal handler can be implemented as signaling operations to the handler task corresponding to the signal. The condition variable cond will be "signaled"[4] when an interrupt entry or handler is installed using the *pthread_cond_signal()* operation and an instance of the associated signal will be generated using *pthread_kill()* when the interrupt entry or handler is to be detached.

This implementation still has a drawback. Handler tasks have to be activated for all the signals during runtime system elaboration, even for signals that will never have handlers, since with dynamic binding we can't know for sure which signals the application may bind to handlers. A task is a fairly heavyweight object to use as a signal handler; it requires a separate stack and at least a partial task control block. This is especially onerous if we provide such a task for each signal.

GNARL solves this problem by introducing a single task, called `Handler_Manager`, that is responsible for taking default actions for all signals and coordinating the attaching and detaching of handlers. This way there is no need to create a special server task for a signal until the application first attaches a handler to it. The basic control structures of the `Handler_Manager` task and an individual signal server, of type `Handler_Task`, are as follows:

```
task body Handler_Manager is
begin
  loop
    select accept Bind_Handler
      (Signal : Ada.Interrupts.Interrupt_ID) do
        pthread_sigmask (SIG_BLOCK,
          signal_set (Signal), old_set);
        -- Mask the given signal for this task so
        -- it is masked in all tasks.
        -- A delivery of the signal will be
        -- caught by the corresp. Handler'Task
        -- while it is waiting on sigwait.
        pthread_cond_signal (cond (Signal));
      end Bind_Handler;
    or accept Unbind_Handler
      (Signal : Ada.Interrupts.Interrupt_ID) do
        -- Currently, there is a handler or an
        -- entry attached and the corresponding
        -- Handler'Task is waiting on sigwait.
        pthread_kill
          (Handler_Task_ID (Signal), Signal);
        -- We have to wake the Handler'Task up
        -- to wait on a condition variable.
        sigaction (Signal,
          default_action (Signal), old_action);
        -- Restore default action of this signal.
        pthread_sigmask (SIG_UNBLOCK,
          signal_set (Signal), old_set);
        -- Unmask the interrupt for this task, to
        -- allow the default action again.
      end Unbind_Handler;
    end select;
  end loop;
end Handler_Manager;

task body Handler_Task is
begin
  loop
    if (not Handler_Installed (Signal)) then
      pthread_mutex_lock (mutex (Signal));
      pthread_cond_wait
        (cond (Signal), mutex (Signal));
      pthread_mutex_unlock (mutex (Signal));
    else
      sig := sigwait (signal_set, Signal);
      if (Handler_Installed (Signal)) then
        <Make a call to an entry or a handler>
        -- Make an entry or protected procedure call
        -- depending on the type of handler.
      end if;
    end if;
  end loop;
end Handler_Task;
```

With the above construct, in order to attach a handler for a signal, the RTS calls the entry `Handler_Manager.-Bind_Handler` after registering the corresponding handler in a global variable. Then `Handler_Manager` adds the signal to its own signal mask, so that all the tasks in the current process will have the signal masked. Then `Handler_Manager` signals an appropriate condition variable so that the `Handler_Task` corresponding to that signal moves from *pthread_cond_wait()* to *sigwait()*. Upon receiving a signal the `Handler_Task` will then wake up and perform appropriate action according to the information stored in the global storage.

[4] Despite the name, this operation, which wakes up a task that is waiting on the condition variable, does not actually involve POSIX signals.

To detach a handler, the RTS calls the entry `Handler_Manager.Unbind_Handler` after registering in global storage that there is no handler attached. Then `Handler_Manager` signals an appropriate `Handler_Task` using *pthread_kill()*. The `Handler_Task` will wake up from the *sigwait()* and move on to the *pthread_cond_wait()*. `Handler_Manager` then installs a default action and removes the signal from its signal mask, so that it is ready to take default action for subsequent deliveries of that signal.

All of the tasks created by the RTS for signal handling are independent of the environment task. Therefore, they are not required to complete before the termination of the environment task. When finalization of environment task is performed these tasks simply go away along with the rest of the POSIX process, as a consequenct of the process exit operation.

4 Performance

The use of protected procedures as signal handlers is more efficient than task entries, since rendezvous requires at least two extra task switches. However, direct use of the `Await_Signal` procedures, which are bindings to *sigwait()*, appear to be the most efficient Ada signal handling mechanism. Both of the other two mechanisms impose the time and memory overhead of creating a runtime system task to act as server.

We suspect that the methods with higher runtime overhead may aggravate a potential for lost signals that is inherent in the POSIX signal semantics. If a signal is generated for a thread while there is already an instance of that signal pending, that signal may be lost; that is, if the thread makes a subsequent call to *sigwait()* or unmasks the signal with *pthread_sigmask()* it may be that only one of the signal occurrences is delivered. Handling signals indirectly though server tasks may increase the probability of missed signals, because the interval during which the signal is masked and unable to be delivered is longer than with a C-style asynchronous handler procedure or the bare *sigwait()*. This is due to the extra overhead of waking up the handler task, making the protected procedure or task entry call (with context switches, in the latter case), and then the continued execution of the handler task while it is looping back around to get ready for the next signal occurrence. However, losing a few signals should not be a serious problem, since correct POSIX applications must be designed to tolerate lost signals.

In this section of the full paper we expect to give performance measurements of the various signal delivery mechanisms supported by GNARL, including measurements of the overhead and statistics on the rates of signal loss. We expect the latter to shed light on whether there is any noticeable increase in the probability of lost signals. The performance results are not ready yet but should be available by the time for the final submission of the paper.

5 Conclusion

GNARL provides several mechanisms for an Ada application to be notified of POSIX signals, including the protected procedure and task entry mechanisms. The exception mechanism is also supported, for the reserved signals. An asynchronous exception mechanism is provided for SIGABRT, this is used to implement task abortion and asynchronous transfer of control. All this is part of the standard GNAT distribution.

GNAT, including GNARL, has been validated for execution on several computers made by Silicon Graphics and running the IRIX[5] operating system. It has been ported to a number of other combinations of hardware and operating systems, including SPARC[6] workstations running the Solaris 2.4[7] operating system. On both IRIX and Solaris 2.4 GNARL makes use of OS kernel threads; this provides true parallel execution on multiprocessor configurations.

The `Await_Signal` interface described here is part of the FSU POSIX.5b prototype implementation. So far, it has only been tested for Solaris 2.4, though it was designed to be simple to port to other UNIX[8]-like systems. In further work, if we are able to obtain funding, we plan to port the POSIX.5b implementation to some of the other systems to which GNARL has been ported.

GNAT is available for free use, by anonymous ftp from `ftp.cs.nyu.edu`. The FSU POSIX.5b implementation will also be made available for free use, by anonymous ftp from `ftp.cs.fsu.edu` (in directory `/pub/PART`). New releases of both systems will continue to appear over time.

More information on GNARL and the POSIX.5b implementation can be found in the web home-page `http://www.cs.fsu.edu/~doh/realtime.html`.

[5]IRIX is a trademark of Silicon Graphics, Inc.
[6]SPARC is a trademark Sun Microsystems, Inc.
[7]Solaris is a trademark of SunSoft, Inc.
[8]UNIX is a trademark of UNIX Systems Laboratories, Inc.

Acknowledgments

This work is part of the FSU POSIX/Ada Real-Time (PART) Project. PART has been funded in part by the HQ U.S. Army CECOM, Software Engineering Directorate, with support from the Ada9X Project Office and Ada Joint Program Office's Ada Technology Insertion Program.

References

[1] ISO/IEC. *ISO/IEC 9945-1: 1990 Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*, 1990. IEEE Std 1003.1-1990.

[2] ISO/IEC. *ISO/IEC 14519-1: 1995, Information Technology— POSIX Ada Language Interfaces—Part 1: Binding for System Application Program Interface (API)*, June 1992. IEEE Std 1003.5-1992.

[3] ISO/IEC. *ISO/IEC 8652: 1995 (E) Information Technology — Programming Languages — Ada*, February 1995.

[4] Richard Kenner. Integrating gnat into gcc. In *TRI-Ada '94 Proceedings*. ACM, 1994.

[5] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 1: Realtime Extension [C Language]*, 1993. IEEE Std 1003.1b-1993.

[6] Technical Committee on Operating Systems and Application Environments of the IEEE. *Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C Language]*, 1995. IEEE Std 1003.1c-1995.

[7] United States Department of Defense. *ANSI/MIL-STD-1815A-1983 Reference Manual for the Ada Programming Language*, February 1983.

[8] New York University. The Gnu NYU Ada Translator (GNAT). Available by anonymous FTP from cs.nyu.edu.