APPLICATION

In order to show how the above features function, we will use an application based on a 68332 processor from Motorola. The ROM monitor is using the serial channel internal to the 68332 to communicate with the host. For this workshop, there will actually be two hosts: a PC running a terminal emulation program and a PC running the XDB C source-level debugger. In this way we can contrast the features of debugging with and without a source-level debugger. The demonstration will show not only how debugging can be accomplished with these tools, but also how to simulate code patches around bugs found in the program using breakpoints.

Debugging Interrupt-Driven Code

In this section we will discuss debugging interrupt driven code and how the monitor may be operated when ISRs (interrupt service routines) are being debugged. The fact that the ROM monitor is running on the actual target allows ISRs to continue to run if desired. In this application we will show how a timer ISR will be allowed to run when the ROM monitor is active and how, alternatively, it may be disabled.

When single stepping through code, an ISR can still be allowed to run if it is of a higher priority than the current ISR being stepped through. This is because when a ISR occurs, the exception process negates the trace bit in the status register. This allows the ISR to execute without tracing.

Additionally, the demonstration will show how tracing can be forced to occur in ISRs by using condition trace points.

After debugging the application and programming it into EPROMs we will discuss how the ROM monitor may be left in the final product so that either a source-level debugger, or a dumb terminal may be connected to a running system to interrupt normal operation in the field.

CONCLUSION

Using a debugging environment which involves embedding the tools into your target offers alternative solutions. As the processor manufacturers continue to design more features into the actual silicon, the use of embedded tools will continue to increase. Processors such as the 68332 which offer a background mode mean that the actual tools take up minimal target resources. The Intel 80386 even offers hardware breakpoint capabilities designed into the actual chip. In the meantime, application developers are looking to combinations of tools to satisfy their programming and debugging needs, especially on large projects where several ROM monitors are used for every emulator.

Hopefully this paper has given you some insight into a ROM monitor's capabilities and limitations, allowing you to determine if this low-cost solution is an alternative to in-circuit emulation.

This paper was co-authored by Andy Lantz of Intermetrics Microsystems Software, Inc., and Peter Dawson of EST Corp.
Understanding POSIX.4 and POSIX.4a

Bill O. Gallmeister
Lynx Real-Time Systems, Inc.
July 23, 1991

1 INTRODUCTION

The IEEE POSIX group is concerned with standardizing a wide range of operating system services, from basic functionality (1003.1) through networking (1003.12), security (1003.6), and real-time (1003.4). The 1003.4 working group is currently balloting two related standards which will inevitably become international standards -- perhaps by the end of 1992. [POSIX.4] is a proposed standard for the support of "applications with real-time requirements". [POSIX.4a] is the "threads extension" to 1003.1. These two standards offer that which has never before been possible in the fragmented world of real-time -- some hope that applications can be written so as to be easily portable from one real-time operating system to the next.

The facilities of 1003.4 (POSIX.4) and 1003.4a (POSIX.4a) are already being demanded by large Federal and commercial customers. Support for these standards will become a powerful tool for addressing these markets. What, though, is in these standards? Will they meet the needs of real-time applications? And, equally important, where do they fall short?

We have recently completed a project to support all of POSIX.4 and POSIX.4a. This was required functionality for the operating system on Space Station Freedom. Our recent release, LynxOS 2.0, incorporates all the features of 1003.4 and 1003.4a. As a result, we have been able to gain some very early experience with these facilities. This paper is an introduction to the facilities of POSIX.4 and POSIX.4a, focusing on the needs of real-time applications, where POSIX meets these needs, and where it does not.

In the next section, we provide a short rationale for the importance of source code portability for real-time applications. After that, the features of POSIX are briefly described. In the section following, we discuss the areas where POSIX does not provide support for real-time applications. A conclusion follows.

2 WHY IS PORTABILITY IMPORTANT?

Real-time applications are long-lived. Real-time applications, aside from being the most demanding applications to program, are also among the longest-lived of all software systems. This is largely because of the very complexity of a real-time application. Software is a surprisingly subtle and difficult discipline, and of late there have been few breakthroughs which reduce the software life cycle or improve the quality of software. On top of this, the software in a real-time application is more difficult than that in a regular application, because the real-time problem domain includes an extra dimension, time. Once this complex software is completed, there is more inertia to it, and the need for re-engineering a working real-time system is going to be very carefully scrutinized.

In contrast, hardware is relatively short-lived. It wears out or is rendered obsolete or unserviceable by ascendant technologies and defunct suppliers. Furthermore, hardware rapidly increases in performance[1]; the market demands for the latest, faster processor can result in amazing market pressure to support new hardware. Thus, we have a fundamental mismatch between the long life of a piece of software and the short life of software. This mismatch implies that software must frequently be ported from one hardware base to another.

3 POSIX FOR APPLICATION PORTABILITY

Decreasing the amount of effort involved in a port is what the POSIX standards are all about -- the source-code portability of applications software. The intention of POSIX is to allow an application, running on an archaic (but POSIX-conformant) system, to be easily recompiled for a cutting-edge (but still POSIX-conformant) system, and minimize the difficulties generally encountered in porting efforts.

POSIX (Portable Operating System Interface) is the colloquial name for the IEEE 1003 committee, which was formed to standardize interfaces to UNIX[2]. UNIX was chosen because of its wide availability and growing popularity. However, UNIX was not originally designed as a real-time operating system. Realizing that real-time applications constituted an important audience for a standard operating system interface, POSIX formed a separate working group to address the special requirements of real-time applications. For additional information regarding the base POSIX standard, see [POSIX.1] and [2].

4 REAL-TIME POSIX: 1003.4 AND 1003.4a

UNIX is not, by origin, a real-time operating system. The major variants, Berkeley UNIX (BSD) and AT&T UNIX (System V), do not provide deterministic I/O, or an appropriate processing model for most real-time applications. However, there is nothing in the basic architecture of UNIX that prevents its being re-implemented for real-time service. In other words, the non-real-time nature of UNIX is not due to the architecture of UNIX, but only to the currently prevalent implementations of UNIX. LynxOS 2.0, a complete re-implementation of UNIX with real-time requirements first and foremost, provides an existence proof for this statement. The task of the 1003.4 group has been to standardize those elements of UNIX that have never been specified (such as the scheduling discipline), and, where necessary, to propose extensions to POSIX where real-time applications will require them. The work of 1003.4 can be grouped into three major areas: the...

[1] UNIX is a trademark of UNIX Systems Laboratory.
Understanding POSIX.4 and POSIX.4a

Bill O. Gallmeister
Lynx Real-Time Systems, Inc.

July 23, 1991

1 INTRODUCTION

The IEEE POSIX group is concerned with standardizing a wide range of operating system services, from basic functionality (1003.1) through networking (1003.12), security (1003.6), and real-time (1003.4). The 1003.4 working group is currently balloting two related standards which will inevitably become international standards -- perhaps by the end of 1992. [POSIX.4] is a proposed standard for the support of "applications with real-time requirements". [POSIX.4a] is the "threads extension" to 1003.1. These two standards offer that which has never before been possible in the fragmented world of real-time -- some hope that applications can be written so as to be easily portable from one real-time operating system to the next.

The facilities of 1003.4 (POSIX.4) and 1003.4a (POSIX.4a) are already being demanded by large Federal and commercial customers. Support for these standards will become a powerful tool for addressing these markets. What, though, is in these standards' ability to meet the needs of real-time applications? And, equally important, where do they fall short?

We have recently completed a project to support all of POSIX.4 and POSIX.4a. This was required functionality for the operating system on Space Station Freedom. Our recent release, LynxOS 2.0, incorporates all the features of 1003.4 and 1003.4a. As a result, we have been able to gain some very early experience with these facilities. This paper is an introduction to the facilities of POSIX.4 and POSIX.4a, focusing on the needs of real-time applications, where POSIX meets these needs, and where it does not.

In the next section, we provide a short rationale for the importance of source-code portability for real-time applications. After that, the features of POSIX are briefly described. In the section following, we discuss the areas where POSIX does not provide support for real-time applications. A conclusion follows.

2 WHY IS PORTABILITY IMPORTANT?

Real-time applications are long-lived. Real-time applications, aside from being the most demanding applications to program, are also among the longest-lived of all software systems. This is largely because of the very complexity of a real-time application. Software is a surprisingly subtle and difficult discipline, and of late there have been few breakthroughs which reduce the software life cycle or improve the quality of software. On top of this, the software in a real-time application is more difficult than that in a regular application, because the real-time problem domain includes an extra dimension, time. Once this complex software is completed, there is more inertia to it, and the need for re-engineering a working real-time system is going to be very carefully scrutinized.

In contrast, hardware is relatively short-lived. It wears out or is rendered obsolete or unserviceable by ascendant technologies and defunct suppliers. Furthermore, hardware rapidly increases in performance[1], the market demands for the latest, faster processor can result in amazing market pressure to support new hardware. Thus, we have a fundamental mismatch between the long life of a piece of software and the short life of software. This mismatch implies that software must frequently be ported from one hardware base to another.

3 POSIX FOR APPLICATION PORTABILITY

Decreasing the amount of effort involved in a port is what the POSIX standards are all about -- the source-code portability of applications software. The intention of POSIX is to allow an application, running on an archaic (but POSIX-conformant) system, to be easily recompiled for a cutting-edge (but still POSIX-conformant) system, and minimize the difficulties generally encountered in porting efforts.

POSIX (Portable Operating System Interface) is the colloquial name for the IEEE 1003 committee, which was formed to standardize interfaces to UNIX. UNIX was chosen because of its wide availability and growing popularity. However, UNIX was not originally designed as a real-time operating system. Realizing that real-time applications constituted an important audience for a standard operating system interface, POSIX formed a separate working group to address the special requirements of real-time applications. For additional information regarding the base POSIX standard, see [POSIX.1] and [2].

4 REAL-TIME POSIX: 1003.4 AND 1003.4a

UNIX is not, by origin, a real-time operating system. The major variants, Berkeley UNIX (BSD) and AT&T UNIX (System V), do not provide deterministic I/O, or an appropriate processing model for most real-time applications. However, there is nothing in the basic architecture of UNIX that prevents its being re-implemented for real-time service. In other words, the non-real-time nature of UNIX is not due to the architecture of UNIX, but only to the currently prevalent implementations of UNIX. LynxOS 2.0, a complete re-implementation of UNIX with real-time requirements first and foremost, provides an existence proof for this statement. The task of the 1003.4 group has been to standardize those elements of UNIX that have never been specified (such as the scheduling discipline), and, where necessary, to propose extensions to POSIX where real-time applications will require them. The work of 1003.4 can be grouped into three major groups: the
processing model, I/O enhancements, and repairs necessary for UNIX to support real-time requirements.

4.1 Processing Model: Scheduling

UNIX, in its standard forms, has no defined scheduler; it is a time-sharing system, and as such, process priorities are constantly adjusted in an effort to maintain the fairness of the system. Real-time applications require, as a first step, absolute control over the processors, meaning a scheduler where the operating system does not modify process priorities.

The 1003.4 proposal defines a standard interface to set the scheduling algorithm and attributes for a process. Atop this interface, standardized scheduling algorithms can be provided. In this way, new scheduling algorithms can be standardized at a later date. 1003.4 itself provides two scheduling disciplines: FIFO, which is a simple preemptive-priority scheduler, and ROUND ROBIN, which adds a quantum to the simple FIFO scheduler.

The scheduler interface is a fairly simple one, which should fit well on most systems that already support a FIFO scheduler. Since this type of scheduler is by far the most common scheduler in real-time operating systems, this interface should require few changes to an existing real-time operating system. We found the necessary modifications to be very simple.

4.2 Processing Model: Time Control

Real-time applications are often composed of separate parts that execute cyclically with various periods. This processing model might be quantified inside another scheduling algorithm, but an interface to time services already existed within Berkeley UNIX. 1003.4 provides a time service that is loosely based on Berkeley interval timers. POSIX timers allow a process to set a timer to go off once, or periodically, at an absolute or a relative time. This interface, a fairly-minimal extension to already existing functionality within UNIX, allows for completely time-driven processing. Since most UNIX systems already support a facility akin to this facility, support of the POSIX.4 requirements is trivial.

4.3 Processing Model: Threads

The UNIX processing model is simple and elegant and, in general, sufficient. However, for real-time applications it is inadequate. The UNIX process is a single flow of control within a protected address space. Sharing data between UNIX processes is difficult and limited. Real-time applications require large numbers of cooperating threads of control, each interacting with the outside world in some way, all cooperating with each other. The essential features of real-time processing are its concurrent nature and its need for massive communication and coordination between concurrent threads. These needs are not addressed well under standard UNIX. 1003.4 extends the standard UNIX processing model with threads: multiple flows of control within a single address space. This change of processing model represents a major change to the way UNIX does business. For this reason, threads have been separated out from the rest of 1003.4 into another proposal, 1003.4a. A reasonable, basic introduction to threads can be found in [8]; descriptions of the machinations necessary for a standard UNIX system to support multiple threads are available in [10, 11].

Threads represent a major change in the way a UNIX system does business. This modification to the system required approximately a man-year and a half of development and another man-year of quality assurance. Interestingly, some real-time kernels may find themselves going the other way. They may already support a threaded model of computation, and will have to expend effort to provide a non-threaded processing model. Furthermore, there may be systems, particularly embedded, MMU-less systems, where support for independent, protected address spaces is impractical.

4.4 Thread and Process Synchronization

The two proposals, 1003.4 and 1003.4a, together satisfy the processing model requirements of real-time applications. The coordination requirements of real-time applications are harder to define: how do the concurrent activities in a real-time application communicate with each other? The short answer is, in many ways [6, 7]. There is a wide body of practice in the industry, and little consensus as to what is “best”. Since threads exist in a common address space, they are more synchronization-intensive than are protected UNIX processes. In either a threaded or a non-threaded environment, it is absolutely critical that the standard synchronization mechanism be primitive and general, so that it can be used as a building block for more sophisticated mechanisms. In addition, the mechanism must be implementable using atomic memory instructions such as test-and-set or compare-and-swap. Such mechanisms allow an implementation requiring less than ten instructions per locking operation. Traps to the operating system for common synchronization operations are absolutely intolerable. 1003.4 has taken the route of providing what was perceived as the lowest common denominator for synchronizing threads and/or processes: the binary semaphore. Using binary semaphores, more advanced synchronization mechanisms can be created. In addition, 1003.4a standardizes mutexes and condition variables, mechanisms that are currently being widely used in existing threaded systems, such as Mach and Topaz.

4.5 I/O in POSIX.4

The standard UNIX model for I/O is simple and elegant, but it does not offer many of the features required for real-time applications. For instance, UNIX files are simply streams of bytes; there is no control over disk geometry. I/O in standard UNIX is synchronous, blocking the calling thread until the I/O is “complete”. I/O is “complete” in normal UNIX when the data has been queued to be written to the relevant device; there is no standard way to force I/O to bypass the disk cache.

4.5.1 Synchronized I/O

POSIX.4 specifies a mechanism whereby an application can dictate that all I/O to a particular file is to be completed in a synchronous fashion. All such I/O is guaranteed to be successfully transferred to the underlying physical medium before the I/O operation completes.
4.1 Processing Model: Scheduling

UNIX, in its standard forms, has no defined scheduler; it is a time-sharing system, and as such, process priorities are constantly adjusted in an effort to maintain the fairness of the system. Real-time applications require, as a first step, absolute control over the processors, meaning a scheduler where the operating system does not modify process priorities.

The 1003.4 proposal defines a standard interface to set the scheduling algorithm and attributes for a process. Atop this interface, standardized scheduling algorithms can be provided. In this way, new scheduling algorithms can be standardized at a later date. 1003.4 itself provides two scheduling disciplines: FIFO, which is a simple preemptive-priority scheduler, and ROUND ROBIN, which adds a quantum to the simple FIFO scheduler.

The scheduler interface is a fairly simple one, which should fit well on most systems that already support a FIFO scheduler. Since this type of scheduler is by far the most common scheduler in real-time operating systems, this interface should require few changes to an existing real-time operating system. We found the necessary modifications to be very simple.

4.2 Processing Model: Time Control

Real-time applications are often composed of separate parts that execute cyclically with various periods. This processing model might be quantified inside another scheduling algorithm, but an interface to time services already existed within Berkeley UNIX. 1003.4 provides a time service that is loosely based on Berkeley interval timers. POSIX timers allow a process to set a timer to go off once, or periodically, at an absolute or a relative time. This interface, a fairly-minimal extension to already existing functionality within UNIX, allows for completely time-driven processing. Since most UNIX systems already support a facility akin to this facility, support of the POSIX.4 requirements is trivial.

4.3 Processing Model: Threads

The UNIX processing model is simple and elegant and, in general, sufficient. However, for real-time applications it is inadequate. The UNIX process is a single flow of control within a protected address space. Sharing data between UNIX processes is difficult and limited. Real-time applications require large numbers of cooperating threads of control, each interacting with the outside world in some way, all cooperating with each other. The essential features of real-time processing are its concurrent nature and its need for massive communication and coordination between concurrent threads. These needs are not addressed well under standard UNIX. 1003.4 extends the standard UNIX processing model with threads: multiple flows of control within a single address space. This change of processing model represents a major change to the way UNIX does business. For this reason, threads have been separated out from the rest of 1003.4 into another proposal, 1003.4a. A reasonable, basic introduction to threads can be found in [8]; descriptions of the machinations necessary for a standard UNIX system to support multiple threads are available in [10, 11].

Threads represent a major change in the way a UNIX system does business. This modification to our system required approximately a man-year and a half of development and another man-year of quality assurance. Interestingly, some real-time kernels may find themselves going the other way. They may already support a threaded model of computation, and will have to expend effort to provide a non-threaded processing model. Furthermore, there may be systems, particularly embedded, MMU-less systems, where support for independent, protected address spaces is impractical.

4.4 Thread and Process Synchronization

The two proposals, 1003.4 and 1003.4a, together satisfy the processing model requirements of real-time applications. The coordination requirements of real-time applications are harder to define: how do the concurrent activities in a real-time application communicate with each other? The short answer is, in many ways[6, 7]. There is a wide body of practice in the industry, and little consensus as to what is "best". Since threads exist in a common address space, they are more synchronization-intensive than are protected UNIX processes. In either a threaded or a non-threaded environment, it is absolutely critical that the standard synchronization mechanism be primitive and general, so that it can be used as a building block for more sophisticated mechanisms. In addition, the mechanism must be implementable using atomic memory instructions such as test-and-set or compare-and-swap. Such mechanisms allow an implementation requiring less than ten instructions per locking operation.

Traps to the operating system for common synchronization operations are absolutely intolerable. 1003.4 has taken the route of providing what was perceived as the lowest common denominator for synchronizing threads and/or processes: the binary semaphore. Using binary semaphores, more advanced synchronization mechanisms can be created. In addition, 1003.4a standardizes mutexes and condition variables, mechanisms that are currently being widely used in existing threaded systems, such as Mach and Topaz.

4.5 I/O in POSIX.4

The standard UNIX model for I/O is simple and elegant, but it does not offer many of the features required for real-time applications. For instance, UNIX files are simply streams of bytes; there is no control over disk geometry. I/O in standard UNIX is synchronous, blocking the calling thread until the I/O is "complete". I/O is "complete" in normal UNIX when the data has been queued to be written to the relevant device; there is no standard way to force I/O to bypass the disk cache.

4.5.1 Synchronized I/O

POSIX.4 specifies a mechanism whereby an application can dictate that all I/O to a particular file is to be completed in a synchronous fashion. All such I/O is guaranteed to be successfully transferred to the underlying physical medium before the I/O operation completes.
We found that the underlying mechanism to perform a synchronous write to disk was already present within our system; given this underlying functionality, synchronized I/O as specified is a simple addition to an existing system.

4.5.2 Asynchronous I/O

POSIX.4 supplies an asynchronous I/O facility. Under standard UNIX, a read or write blocks the calling thread until the I/O is complete. The asynchronous I/O facility allows the calling thread to continue on with its work as soon as the I/O is queued up for completion. Upon actual completion, the thread is asynchronously notified via a POSIX signal.

Given the presence of threads, asynchronous I/O is fairly simple: a new thread can be created to asynchronously perform the I/O. However, even given threads, there are some subtleties to be addressed, especially if higher performance is desired than can be achieved with the simple, intuitive implementation of asynchronous I/O using threads. If the system does not have threads already present, then a different method for asynchronously performing I/O must be devised\[12\]. In addition, the presence of an asynchronous I/O facility may, like DMA, lull the applications programmer into believing that the I/O is somehow taking place magically, with zero overhead. This is in general not the case. At the very least, there is cycle-stealing going on as a separate, dedicated processor reads or writes the data. This overhead must be taken into account when using asynchronous I/O in a real-time system.

4.5.3 Control over File Attributes

Standard UNIX files have no attributes related to the file structure. There is no facility for pre-allocating disk blocks, for instance, or for laying the blocks out contiguously. POSIX.4 provides a mechanism for creating such files, called “real-time” files.

There are many different sorts of facilities that a system may provide for control over disk file characteristics. Not all systems will provide all options. Thus, the real-time file chapter provides an interface whereby the application can query the system as to whether a particular attribute is supported for a particular file or a particular file system.

Direct Device Access: I/O in UNIX is buffered in the operating system. This improves performance in general, but real-time applications sometimes must be able to bypass the buffer cache. The real-time files chapter also provides a mechanism for performing direct I/O to a file — that is, I/O directly from the process address space to the device, without unnecessary intervening buffering.

LynxOS already supported contiguous, pre-allocated disk files; we found it fairly simple to support the POSIX.4 real-time files interface atop our existing kernel facility.

4.6 Repairing UNIX for Real-Time

In addition to the major functional enhancements described above, the POSIX.4 provides a number of additional facilities that are less intrusive in nature.

4.6.1 Shared Memory

Most currently-available UNIX systems already provide a facility for processes to share memory. When threads are not supported (and very few currently existing UNIX systems do), shared memory is essential so that processes can communicate in a high-bandwidth fashion. POSIX.4 specifies a facility that is similar to the mmap interface found in Berkeley-derived systems, and in AT&T’s System V, Release 4\[3\]. Providing the POSIX interface can be done without kernel impact at all on these systems.

4.6.2 Memory Locking

UNIX is, in most of its incarnations, a swapping, paging system. An average application in such a system may have its physical memory context paged or swapped out to disk when other processes need the memory. In a real-time application this is unacceptable; processes must be able to lock their physical context into physical memory or their dispatch latency may grow by orders of magnitude. Modern UNIX systems already provide memory locking facilities; POSIX.4 standardizes these facilities. Providing the POSIX interface can be done without kernel impact at all on these systems.

4.6.3 Enhancements to UNIX Signals

UNIX signals are a mechanism whereby a process or thread can be asynchronously notified of some occurrence, much like a hardware interrupt. However, UNIX signals are unreliable. They are in general not queued for delivery, but instead, are registered by setting a flag in the target thread. If multiple signals occur, many of them can be lost due to this non-queued nature. POSIX.4 has proposed an additional set of signals which shall be queued rather than unreliablely registered.

Extending the signal mechanism was difficult. We spent a good six man-months on the effort.

4.6.4 Message Passing

Standard UNIX provides a few message-passing mechanisms, notably the pipe and the socket. Pipes provide raw streams of bytes; the messages are unstructured. Sockets provide more structured messages, support different protocols, and are network-extensible.

Initially, in looking at the requirements a real-time system might have for a message-passing interface, the POSIX.4 working group determined that neither pipes nor sockets would be suitable for real-time needs. Of the two interfaces, sockets was close to what was required; however, another working group was working on a standardized sockets interface, and their proposal is still under development. The POSIX.4 working group did not want to wait for this proposal, and then extend it for real-time use. Instead, a new message-passing interface was proposed which satisfied all perceived requirements for message passing.

This message-passing interface is large and baroque, and it ran into immediate problems in the balloting process. Our experience in implementing it revealed an interface that is not particularly amenable to high-performance IPC. The mechanism was capable of
We found that the underlying mechanism to perform a synchronous write to disk was already present within our system; given this underlying functionality, synchronized I/O as specified is a simple addition to an existing system.

4.5.2 Asynchronous I/O

POSIX.4 supplies an asynchronous I/O facility. Under standard UNIX, a read or write blocks the calling thread until the I/O is complete. The asynchronous I/O facility allows the calling thread to continue on with its work as soon as the I/O is queued up for completion. Upon actual completion, the thread is asynchronously notified via a POSIX signal.

Given the presence of threads, asynchronous I/O is fairly simple: a new thread can be created to asynchronously perform the I/O. However, even given threads, there are some subtleties to be addressed, especially if higher performance is desired than can be achieved with the simple, intuitive implementation of asynchronous I/O using threads. If the system does not have threads already present, then a different method for asynchronously performing I/O must be devised.[12] In addition, the presence of an asynchronous I/O facility may, like DMA, baffle the applications programmer into believing that the I/O is somehow taking place magically, with zero overhead. This is in general not the case. At the very least, there is cycle-stealing going on as a separate, dedicated processor reads or writes the data. This overhead must be taken into account when using asynchronous I/O in a real-time system.

4.5.3 Control over File Attributes

Standard UNIX files have no attributes related to the file structure. There is no facility for pre-allocating disk blocks, for instance, or for laying the blocks out contiguously. POSIX.4 provides a mechanism for creating such files, called "real-time" files.

There are many different sorts of facilities that a system may provide for control over disk file characteristics. Not all systems will provide all options. Thus, the real-time file chapter provides an interface whereby the application can query the system as to whether a particular attribute is supported for a particular file or a particular file system.

Direct Device Access: I/O in UNIX is buffered in the operating system. This improves performance in general, but real-time applications sometimes must be able to bypass the buffer cache. The real-time files chapter also provides a mechanism for performing direct I/O to a file - that is, I/O directly from the process address space to the device, without unnecessary intervening buffering.

LynxOS already supported contiguous, pre-allocated disk files; we found it fairly simple to support the POSIX.4 real-time files interface atop our existing kernel facility.

4.6 Repairing UNIX for Real-Time

In addition to the major functional enhancements described above, the POSIX.4 provides a number of additional facilities that are less intrusive in nature.

4.6.1 Shared Memory

Most currently-available UNIX systems already provide a facility for processes to share memory. When threads are not supported (and very few currently existing UNIX systems do), shared memory is essential so that processes can communicate in a high-bandwidth fashion. POSIX.4 specifies a facility that is similar to the mmap interface found in Berkeley-derived systems, and in AT&T's System V, Release 4[3]. Providing the POSIX interface can be done without kernel impact at all on these systems.

4.6.2 Memory Locking

UNIX is, in most of its incarnations, a swapping, paging system. An average application in such a system may have its physical memory context paged or swapped out to disk when other processes need the memory. In a real-time application this is unacceptable; processes must be able to lock their physical context into physical memory or their dispatch latency may grow by orders of magnitude. Modern UNIX systems already provide memory locking facilities; POSIX.4 standardizes these facilities. Providing the POSIX interface can be done without kernel impact at all on these systems.

4.6.3 Enhancements to UNIX Signals

UNIX signals are a mechanism whereby a process or thread can be asynchronously notified of some occurrence, much like a hardware interrupt. However, UNIX signals are unreliable. They are in general not queued for delivery, but instead, are registered by setting a flag in the target thread. If multiple signals occur, many of them can be lost due to this non-queued nature. POSIX.4 has proposed an additional set of signals which shall be queued rather than unreliably registered.

Extending the signal mechanism was difficult. We spent a good six man-months on the effort.

4.6.4 Message Passing

Standard UNIX provides a few message-passing mechanisms, notably the pipe and the socket. Pipes provide raw streams of bytes; the messages are unstructured. Sockets provide more structured messages, support different protocols, and are network-extensible.

Initially, in looking at the requirements a real-time system might have for a message-passing interface, the POSIX.4 working group determined that neither pipes nor sockets would be suitable for real-time needs. Of the two interfaces, sockets was close to what was required; however, another working group was working on a standardized sockets interface, and their proposal is still under development. The POSIX.4 working group did not want to wait for this proposal, and then extend it for real-time use. Instead, a new message-passing interface was proposed which satisfied all perceived requirements for message passing.

This message-passing interface is large and baroque, and it ran into immediate problems in the balloting process. Our experience in implementing it revealed an interface that is not particularly amenable to high-performance IPC. The mechanism was capable of...
The charter of the POSIX working group leaves out a number of areas which are important for portability:

- **Binary Compatibility**: As standards explicitly for the source-code portability of applications, the POSIX standards do not cover any sort of binary compatibility, as found in the System V Application Binary Interface documents or the binary compatibility standards put forward by various industry groups. Binary compatibility is the ability to run compiled programs on disparate operating systems and achieve identical results. This sort of compatibility (or perhaps its cousin, ANDF (Architecture Neutral Distribution Format)) is critical if so-called "shrink-wrapped", off-the-shelf real-time applications are to be supported[3, 4, 5].

- **Device Specific Functions**: The task of POSIX was to standardize existing practice, not to innovate. In certain realms, such as the area of direct application control of hardware devices, there is very little existing practice to standardize. In this case, the working groups opted to standardize nothing rather than to provide a (possibly misleading) glimmer of hope that interfaces to distinct hardware devices might somehow be made standard.

- **A Complete Suite of Performance Metrics**: The task of POSIX was to standardize existing practice, not to innovate. In certain realms, such as the area of direct application control of hardware devices, there is very little existing practice to standardize. In this case, the working groups opted to standardize nothing rather than to provide a (possibly misleading) glimmer of hope that interfaces to distinct hardware devices might somehow be made standard.

1. The performance metrics are optional. This means that it is up to the systems procurer to demand the standard metrics. This is only a small step forward from the current state of affairs, where each real-time vendor offers its own performance numbers, which are absolutely unrelated to everyone else's. Worse, a vendor may be able to avoid reporting performance numbers altogether; this might lead to degenerate implementations. For instance, typical existing UNIX systems could be fairly easily modified to support a FIFO scheduler as required by POSIX 4; however, without a preemptible kernel and properly written drivers, the worst-case dispatch latency for the system could easily be upwards of a second. Obviously, in such a case the presence of a "real-time" scheduler is little more than a ruse.

2. The performance metrics do not cover 1003.1. The POSIX 4 and POSIX 4a proposals are extensions to the base POSIX standard, 1003.1. There are no performance metrics for 1003.1, because metrics were not as critical to the 1003.1 working group. So, even if the 1003.4 and 1003.4a metrics are provided, they cover only a fraction of the standard-mandated functionality.

### 5 WHAT POSIX DOES NOT DO

**The performance metrics do not cover 1003.1. The POSIX 4 and POSIX 4a proposals are extensions to the base POSIX standard, 1003.1. There are no performance metrics for 1003.1, because metrics were not as critical to the 1003.1 working group. So, even if the 1003.4 and 1003.4a metrics are provided, they cover only a fraction of the standard-mandated functionality.**

### 6 USING POSIX NOW

The POSIX 4 and POSIX 4a facilities are not yet approved standards. They are currently being balloted. At the conclusion of balloting, POSIX 4 and POSIX 4a will become official IEEE standards; shortly thereafter, they will become ANSI/ISO standards. These events will probably happen by the end of 1992.

Meanwhile, it is not too early to begin using the POSIX facilities for real-time. Products such as LynxOS 2.0 already support early versions of the POSIX facilities; and while there are some changes ongoing in response to balloting objections, these changes are not sweeping from the standpoint of the application developer. The exception to this statement is the message-passing facility, which is being changed radically. Most changes visible to the application developer today will be in the nature of renaming system interface calls, rearranging parameters and renaming types. The basic structure of POSIX 4 and POSIX 4a seems solid; using the facilities now will bring an application very close to the final standard.

### 7 CONCLUSION

Portability of real-time applications is becoming a major concern in the Federal and commercial marketplaces. Vendors of applications will require increasingly portable software in order to exploit a continuing stream of new, cheap, faster hardware; this, in turn, is already resulting in demands for standardized facilities for real-time application support.

The facilities of POSIX 4 and POSIX 4a provide a necessary basis for standardized operating system support for real-time applications. In our implementation, we found that the facilities, for the most part, fit well over an existing "real-time UNIX". That is, if the system already supports UNIX functionality, as well as required facilities for real-time, modifications to support POSIX 4 are small. However, there are a few sticky points, notably the support of POSIX 4a threads, the POSIX 4 extended signals, and POSIX 4 IPC message passing. From the other side, the facilities of POSIX provide a reasonable base set of functions for real-time applications. The early feedback from our customers has been positive in this regard.

However, the mere fact that a system is POSIX 4-conformant is not sufficient to guarantee suitability for use in a real-time application. LynxOS was a system capable of hard real-time performance before the project to implement POSIX 4 and POSIX 4a, and LynxOS is still capable of hard real-time performance. However, as indicated above, many of the features described by POSIX 4 and POSIX 4a can be easily implemented atop existing, non-real-time UNIX systems. Implementing the POSIX interfaces will not make these systems any more capable of real-time performance. Procurers of real-time operating systems must be careful to insist on the provision of the standard performance metrics so that they can be sure they are comparing like numbers, and also to determine whether the system is, in fact, capable of providing real-time performance that matches its real-time functions.
performance roughly equivalent to that of System V messages. Work is currently ongoing in 1003.4 to trim back the message-passing proposal to a facility that is simpler and more efficient. In addition, the real-time requirements have been submitted to the networking group of POSIX for their consideration and possible inclusion into a real-time extension to the sockets facility.

5 WHAT POSIX DOES NOT DO

The charter of the POSIX working group leaves out a number of areas which are important for portability:

- **Binary Compatibility:** As standards explicitly for the source-code portability of applications, the POSIX standards do not cover any sort of binary compatibility, as found in the System V Application Binary Interface documents or the binary compatibility standards put forward by various industry groups. Binary compatibility is the ability to run compiled programs on disparate operating systems and achieve identical results. This sort of compatibility (or perhaps its cousin, ANDF [Architecture Neutral Distribution Format]) is critical if so-called "shrink-wrapped", off-the-shelf real-time applications are to be supported[3, 4, 5].

- **Device Specific Functions:** The task of POSIX was to standardize existing practice, not to innovate. In certain realms, such as the area of direct application control of hardware devices, there is very little existing practice to standardize. In this case, the working groups opted to standardize nothing rather than to provide a (possibly misleading) glimmer of hope that interfaces to distinct hardware devices might somehow be made standard.

- **A Complete Suite of Performance Metrics:** The POSIX.4 working group realized the importance of standardized performance metrics to the procurement of POSIX-conformant real-time operating systems. The working group has proposed a fairly extensive set of performance measurements which the system may provide. There are a few problems with the approach currently found in POSIX.4 and POSIX.4a.

1. The performance metrics are optional. This means that it is up to the systems procurer to demand the standard metrics. This is only a small step forward from the current state of affairs, where each real-time vendor devises and quotes its own performance numbers, which are absolutely unrelated to everyone else's. Worse, a vendor may be able to avoid reporting performance numbers altogether; this might lead to degenerate implementations. For instance, typical existing UNIX systems could be fairly easily modified to support a FIFO scheduler as required by POSIX.4; however, without a preemptible kernel and properly written drivers, the worst-case dispatch latency for the system could easily be upwards of a second. Obviously, in such a case the presence of a "real-time" scheduler is little more than a ruse.

2. The performance metrics do not cover 1003.1. The POSIX.4 and POSIX.4a proposals are extensions to the base POSIX standard, 1003.1. There are no performance metrics for 1003.1, because metrics were not as critical to the 1003.1 working group. So, even if the 1003.4 and 1003.4a metrics are provided, they cover only a fraction of the standard-mandated functionality.

6 USING POSIX NOW

The POSIX.4 and POSIX.4a facilities are not yet approved standards. They are currently being balloted. At the conclusion of balloting, POSIX.4 and POSIX.4a will become official IEEE standards; shortly thereafter, they will become ANSI/ISO standards. These events will probably happen by the end of 1992.

Meanwhile, it is not too early to begin using the POSIX facilities for real-time. Products such as LynxOS 2.0 already support early versions of the POSIX facilities; and while there are some changes ongoing in response to balloting objections, these changes are not sweeping from the standpoint of the application developer. The exception to this statement is the message-passing facility, which is being changed radically. Most changes visible to the application developer today will be in the nature of renaming system interface calls, rearranging parameters and renaming types. The basic structure of POSIX.4 and POSIX.4a seems solid; using the facilities now will bring an application very close to the final standard.

7 CONCLUSION

Portability of real-time applications is becoming a major concern in the Federal and commercial marketplaces. Vendors of applications will require increasingly portable software in order to exploit a continuing stream of new, cheap, faster hardware; this, in turn, is already resulting in demands for standardized facilities for real-time application support.

The facilities of POSIX.4 and POSIX.4a provide a necessary basis for standardized operating system support for real-time applications. In our implementation, we found that the facilities, for the most part, fit well over an existing "real-time UNIX". That is, if the system already supports UNIX functionality, as well as required facilities for real-time, modifications to support POSIX.4 are small. However, there are a few sticky points, notably the support of POSIX.4a threads, the POSIX.4 extended signals, and POSIX.4 IPe message passing. From the other side, the facilities of POSIX provide a reasonable base set of functions for real-time applications. The early feedback from our customers has been positive in this regard.

However, the mere fact that a system is POSIX.4-conformant is not sufficient to guarantee suitability for use in a real-time application. LynxOS was a system capable of hard real-time performance before the project to implement POSIX.4 and POSIX.4a, and LynxOS is still capable of hard real-time performance. However, as indicated above, many of the features described by POSIX.4 and POSIX.4a can be easily implemented atop existing, non-real-time UNIX systems. Implementing the POSIX interfaces will not make these systems any more capable of real-time performance. Procurers of real-time operating systems must be careful to insist on the provision of the standard performance metrics so that they can be sure they are comparing like numbers, and also to determine whether the system is, in fact, capable of providing real-time performance that matches its real-time functions.
References


BUILDING PROTECTED MODE EMBEDDED SYSTEMS

Jack Ganssle
Softaid, Inc.
Columbia, MD

Jack Ganssle is president of Softaid Inc., a vendor of microprocessor development tools. He is also a columnist for Embedded Systems Programming and a contributing editor of Ocean Navigator magazine. He has been developing embedded microprocessor products since 1973.