Bill Gallmeister
Lynx Real-Time Systems
Los Gatos, Calif.

Bill Gallmeister is a software engineer at Lynx Real-Time Systems and vice-chair of the IEEE POSIX 1003.4 working group. Aside from in-depth technical experience with the material at hand, he has extensive experience leading small and large groups, and has published papers and given talks on the POSIX proposals.
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 three related standards which will inevitably become international standards. POSIX.4 (Realtime Extensions for Portable Operating Systems, Draft 12, 1992) is a proposed standard for the support of "applications with real-time requirements". POSIX.4a (Threads Extensions for Portable Operating Systems, Draft 6, 1992) 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. In addition, a profile document, POSIX.13 (POSIX Realtime Application Support (AES), Draft 5, 1992) entered balloting in early 1992. The four profiles in 1003.13 enumerate functionality from the various POSIX standards that must be present in order to support applications from embedded systems up to workstation or supercomputer real-time applications. Additionally, the 1003.4 working group has commenced work on POSIX 1003.4b, a new set of functions that provides more real-time functionality, including access to interrupt vectors, sporadic server scheduling and access to typed memory.

The facilities of Real-Time POSIX 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? How do they meet the needs of real-time applications? And, equally important, where do they fall short?

LynxOS, starting with version 2.0, incorporates all the features of early drafts of 1003.4 and 1003.4a. As the earliest adopter of Real-Time POSIX, we have been able to gain some very early experience with these facilities. This paper is an introduction and an update to the functionality, including access to interrupt vectors, sporadic server scheduling and access to typed memory.

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.

POSIX FOR APPLICATION PORTABILITY

Decreasing the amount of effort involved in an application 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 arbitrary (but POSIX-conformant) system, to be easily moved over to another arbitrary (but still POSIX-conformant) system, and minimize the difficulties generally encountered in porting efforts. This ease of portability enables an application vendor to preserve software capital, even in the face of rapidly changing hardware bases.

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 cite{POSIX1} and cite{LEWINE91}.

UNIX, on which POSIX is based, is not generally thought of as a real-time operating system. The major variants, Berkeley UNIX (BSD) and AT&T UNIX (System V), provide few of the features, and none of the performance, generally regarded as crucial for the support of real-time applications. Typical UNIX implementations do not provide deterministic scheduling, 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, a complete re-implemention of UNIX with real-time requirements first and foremost, provides all the functionality of standard UNIX operating systems together with the deterministic high-performance of proprietary real-time kernels.

The task of the 1003.4 group has been to standardize the facilities necessary for the support of real-time under a UNIX system. This job includes standardizing those elements of UNIX that have never been specified (such as the scheduling discipline), and, where necessary, proposing extensions to standard UNIX 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.

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.

1UNIX is a trademark of UNIX Systems Laboratory.
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 three related standards which will inevitably become international standards. POSIX.4 (Realtime Extensions for Portable Operating Systems, Draft 12, 1992) is a proposed standard for the support of "applications with real-time requirements". POSIX.4a (Threads Extensions for Portable Operating Systems, Draft 6, 1992) 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. In addition, a profile document, POSIX.13 (POSIX Realtime Application Support (AEP), Draft 5, 1992) entered balloting in early 1992. The four profiles in 1003.13 enumerate functionality from the various POSIX standards that must be present in order to support applications from embedded systems up to workstation or supercomputer real-time applications. Additionally, the 1003.4 working group has commenced work on POSIX 1003.4b, a new set of functions that provides more real-time functionality, including access to interrupt vectors, sporadic server scheduling and access to typed memory.

The facilities of Real-Time POSIX 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? How do they meet the needs of real-time applications? And, equally important, where do they fall short?

LynxOS, starting with version 2.0, incorporates all the features of early drafts of 1003.4 and 1003.4a. As the earliest adopter of Real-Time POSIX, we have been able to gain some very early experience with these facilities. This paper is an introduction and an update to the enhancements, and repairs necessary for UNIX to support real-time requirements. 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, a complete re-implementation of UNIX with real-time requirements first and foremost, provides all the functionality of standard UNIX operating systems together with the deterministic high-performance of proprietary real-time kernels.

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.

POSIX FOR APPLICATION PORTABILITY

Decreasing the amount of effort involved in an application 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 arbitrary (but POSIX-conformant) system, to be easily moved over to another arbitrary (but still POSIX-conformant) system, and minimize the difficulties generally encountered in porting efforts. This ease of portability enables an application vendor to preserve software capital, even in the face of rapidly changing hardware bases.

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 "cite{POSIX1}" and "cite{LEWINE91}.

UNIX, on which POSIX is based, is not generally thought of as a real-time operating system. The major variants, Berkeley UNIX (BSD) and AT&T UNIX (System V), provide few of the features, and none of the performance, generally regarded as crucial for the support of real-time applications. Typical UNIX implementations do not provide deterministic scheduling, 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, a complete re-implementation of UNIX with real-time requirements first and foremost, provides all the functionality of standard UNIX operating systems together with the deterministic high-performance of proprietary real-time kernels.

The task of the 1003.4 group has been to standardize the facilities necessary for the support of real-time under a UNIX system. This job includes standardizing those elements of UNIX that have never been specified (such as the scheduling discipline), and, where necessary, proposing extensions to standard UNIX 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.

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. Atoh 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.
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.

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. 1003.4a extends the standard UNIX processing model with threads: multiple flows of control within a single address space. Threads are similar to the tasking model used in Ada, and in fact, supporting threads under UNIX is subject to the same difficulties as supporting Ada tasking under UNIX. 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 \cite{MCJONES87,JONES91}; descriptions of the machinations necessary for a standard UNIX system to support multiple threads are available in \cite{BOG91}; descriptions of the standard synchronization mechanism be primitive and general, so that common denominator for synchronizing threads and/or processes: the counting semaphore.

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\cite{SCHWAN88,AXFORD90}. 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 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 on the order of ten instructions per locking operation. Full operating system calls for common synchronization operations are generally intolerable. 1003.4 has taken the route of providing what was perceived as a lowest common denominator for synchronizing threads and/or processes: the counting semaphore.

Using counting 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 OSF/1.

Note that POSIX.4, as initially proposed, specified binary semaphores as a synchronization mechanism. Counting semaphores have taken the place of binary semaphores because they are more general, are generally as simple to support as binary semaphores, and the typical usage of a binary semaphore is that of a subset of the counting semaphore.

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.

Synchronized I/O: To circumvent the normal UNIX buffer cache, 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.

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.

Asynchronous I/O can be easily supported if the underlying system provides threads. If the system does not have threads already present, then a different method for asynchronously performing I/O must be devised\cite{BUCK91}. In any event, remember that the I/O still requires processing. The presence of an asynchronous I/O facility may, like DMA, lull the applications programmer into believing that the I/O is taking place with zero overhead. This is usually 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. Generally, using threads to perform asynchronous I/O will provide a more comprehensible and deterministic picture of what is going on in a real-time application. Nevertheless, asynchronous I/O has its place and has been widely used in real-time and supercomputer applications. Indeed, given enough hardware support (dedicated I/O processors, dual ported memory, and so forth), asynchronous I/O may be supported with little run-time overhead.

Control over File Attributes: Standard UNIX files have no attributes related to the file.
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.

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. 1003.4a extends the standard UNIX processing model with threads: multiple flows of control within a single address space. Threads are similar to the tasking model used in Ada, and in fact, supporting threads under UNIX is subject to the same difficulties as supporting Ada tasking under UNIX. 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 [MCJONES87, JONES91]; descriptions of the machinations necessary for a standard UNIX system to support multiple threads are available in [BOG91]; descriptions of the common denominator for synchronizing threads and/or processes: the counting semaphore.

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 [SCHWAN88, AXFORD90]. 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 on the order of ten instructions per locking operation. Full operating system calls for common synchronization operations are generally intolerable. 1003.4 has taken the route of providing what was perceived as a lowest common denominator for synchronizing threads and/or processes: the counting semaphore.

Using counting 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 OSF/1.

Note that POSIX.4, as initially proposed, specified binary semaphores as a synchronization mechanism. Counting semaphores have taken the place of binary semaphores because they are more general, are generally as simple to support as binary semaphores, and the typical usage of a binary semaphore is a subset of the counting semaphore.

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.

Synchronized I/O: To circumvent the normal UNIX buffer cache, 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.

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.

Asynchronous I/O can be easily supported if the underlying system provides threads. If the system does not have threads already present, then a different method for asynchronously performing I/O must be devised [BUCK91]. In any event, remember that the I/O still requires processing. The presence of an asynchronous I/O facility may, like DMA, lull the applications programmer into believing that the I/O is taking place with zero overhead. This is usually 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. Generally, using threads to perform synchronous I/O will provide a more comprehensible and deterministic picture of what is going on in a real-time application. Nevertheless, asynchronous I/O has its place and has been widely used in real-time and supercomputer applications. Indeed, given enough hardware support (dedicated I/O processors, dual ported memory, and so forth), asynchronous I/O may be supported with little run-time overhead.

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 files 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 throughput 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. In addition, POSIX.4 provides mmap, a function from existing UNIX systems that can be used to directly map device registers into a process’ address space for the purpose of directly performing device operations. Note, however, that this facility is not a complete solution, since most I/O devices generally interrupt, and there is not yet any standard way of gaining access to interrupt vectors. 1003.4b, as mentioned above, is attempting to standardize application access to interrupts, but such work is a long way from being standardized.

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.

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 support threads), shared memory is essential for high-bandwidth communication. 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 4e/v[SVR4]. In addition, in response to balloting, POSIX.4 has now standardized the mmap call itself, providing at least an interface for generalized file mapping (POSIX.4 only guarantees that mmap will work with shared memory objects, not necessarily with disk files, devices, or anything else).

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. Note that on embedded, diskless systems there is no real need to lock memory down—it isn’t going anywhere. However, for portability to a larger, possibly demand-paged UNIX system, an application should always lock down memory it does not want paged—even if that operation is a no-op on the target hardware du jour!

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.

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.

As reported earlier, POSIX.4’s initial cut at a message-passing facility was far too complex and heavyweight to survive balloting. The message-passing proposal has now been cut back to a facility that is simpler and more efficient. This facility has been much more successful in subsequent rounds of balloting. In addition, the networking group of POSIX is currently working on real-time extensions to the socket facilities.

WHAT POSIX DOES NOT DO

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

1. 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. (See SVR4,MOTOBCS, GART90).

2. 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.

Recent work on 1003.4b has provided drafts of functionality for direct application access to interrupts, and for a general “do anything to a driver” facility that is modeled on the ioctl call that all UNIX systems support. As mentioned above, this work will not be balloted for some time; it is far from being standardized.

3. 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
structure. There is no facility for pre-allocating disk blocks, for instance, or for laying the blocks out contiguously. POSIXA 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 files 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 throughput 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. In addition, POSIXA provides mmap, a function from existing UNIX systems that can be used to directly map device registers into a process’ address space for the purpose of directly performing device operations. Note, however, that this facility is not a complete solution, since most I/O devices generally interrupt, and there is not yet any standard way of gaining access to interrupt vectors. l003Ab, as mentioned above, is attempting to standardize application access to interrupts, but such work is a long way from being standardized.

Repairing UNIX for Real-Time

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

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 support threads), shared memory is essential for high-bandwidth communication. POSIXA specifies a facility that is similar to the mmap interface found in Berkeley-derived systems, and in AT&T’s System V, Release 4Cite{SVR4}. In addition, in response to ballooning, POSIXA has now standardized the mmap call itself, providing at least an interface for generalized file mapping (POSIXA only guarantees that mmap will work with shared memory objects, not necessarily with disk files, devices, or anything else).

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; POSIXA standardizes these facilities. Note that on embedded, diskless systems there is no real need to lock memory down—it isn’t going anywhere. However, for portability to a larger, possibly demand-paged POSIX system, an application should always lock down memory it does not want paged—even if that operation is a no-op on the target hardware du jour!

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. POSIXA has proposed an additional set of signals which shall be queued rather than unreliably registered.

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-extendible.

As reported earlier, POSIXA’s initial cut at a message-passing facility was far too complex and heavyweight to survive balloting. The message-passing proposal has now been cut back to a facility that is simpler and more efficient. This facility has been much more successful in subsequent rounds of balloting. In addition, the networking group of POSIX is currently working on real-time extensions to the socket facilities.

WHAT POSIX DOES NOT DO

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

1. Binary Compatibility As standards explicitly for the source-code portability of applications, the POSIXA 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 portable:

Recent work on 1003.4b has provided drafts of functionality for direct application access to interrupts, and for a general “do anything to a driver” facility that is modeled on the ioctl call that all UNIX systems support. As mentioned above, this work will not be balloted for some time; its far from being standardized.

2. 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 POSIXA 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.

4. 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 preemtable 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.

5. 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.

POSIX XLROFILES AND CONFORMANCE

The POSIX standards all eventually feed into a single large international standard, ISO 9945. This standard will, over time, grow to encompass real-time, security, windowing systems, networking, and so forth. ISO 9945 is, understandably, going to become a very large set of requirements. It is doubtful that any vendor will be able to fulfill all the requirements of ISO 9945—that standard addresses far too many diverse interests. In addition, even the features of POSIX.1 are too much for small, embedded kernels. These embedded systems have never been able to support even the most basic of UNIX functionality, the fork system call, which duplicates a process context. Yet there is much of POSIX.4 and POSIX.4a that is of crucial interest to applications writers. How will these people be able to determine which systems meet their needs for real-time functionality? How will they be able to avoid being deceived by specious claims of POSIX conformance, which may be backed up by little more than three or four POSIX calls?

The answer lies in POSIX profiles. A profile can be looked at in two ways. From the system vendor’s point of view, it specifies what a particular application area is really interested in. From the application writer’s point of view, it identifies the facilities that a system must have in order to be useful to the application. A profile is little more than a set of required functionality called out from base standards. For instance, POSIX 1003.13 contains a "minimal real-time system" profile. This profile includes the threads of 1003.4a, priority scheduling, timers, POSIX.4 I/O enhancements and message passing (among other things). These facilities are pretty much what embedded applications running in small, MMU-less environments need. The profile calls out fairly little of the POSIX 1003.1 standard, since a lot of it is irrelevant to small, embedded systems.

POSIX 1003.13 includes four profiles. At the low end is the aforementioned minimal profile: at the high end is a multi-purpose, "kitchen sink" real-time profile that includes just about all of POSIX.4, POSIX.4a, POSIX.1, and POSIX.2. In between are two medium-sized profiles based on avionics practice (the dedicated-purpose profile) and on embedded systems slightly larger than the minimal (the controller profile).

What is "POSIXness"?

Given that there can be POSIX systems as small as a minimal real-time system or as big as a supercomputer, it is important to have a good handle on exactly what one is getting in a system incorporating POSIX features. Many think of POSIX as being a UNIX standard; it is possible that such people will be tricked into thinking that any system associating itself with the word POSIX has a full complement of UNIX features. This is not so. A UNIX-like system must contain, at a minimum, the full complement of POSIX.1 features (the ability to run multiple processes is crucial, and most minimal real-time kernels will not be able to do this), the full set of POSIX.2 commands (the UNIX command interpreter shell and a significant portion of the usual UNIX commands), and, of course, the ability to write POSIX-conformant applications (a language binding, probably either C or Ada).

USING POSIX NOW

The POSIX.4, POSIX.4a, and POSIX.13 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. POSIX.4 will probably be approved by the IEEE by the end of 1992, with POSIX.4a perhaps a year behind it. POSIX.13 cannot become an official standard until the base standards it relies on (POSIX.4 and POSIX.4a) become finalized. Thus, POSIX.13 will not be signed, sealed, and delivered until some time in 1994.

Meanwhile, it is not too early to begin using the POSIX facilities for real-time. Products such as LynxOS already support early versions of the POSIX facilities; and while there are some changes ongoing in response to ballotting objections, these changes are not sweeping from the standpoint of the application developer. Exceptions to this statement are the message-passing facility, which was pruned back substantially, and binary semaphores, which became counting semaphores. These changes are not as radical as one might think. First, message passing has had many baroque options removed. The basic functionality—send a message, receive a message—has not changed; and that functionality is all that the majority of applications will use. As for semaphores, counting semaphores are a generalization of binary semaphores; applications coded to use binary semaphores correctly will be able to switch over to counting semaphores with no change in program logic. Most changes visible to the application developer today will be in the nature of renaming system interface calls, rearranging parameters and re-naming 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.

However, POSIX.13 is a thin, simple document, little more than a set of chapters from POSIX 4 and POSIX.4a; it will be "final" in all but name, long before POSIX.4a is finalized.
POSIX PROFILES AND CONFORMANCE

The POSIX standards all eventually feed into a single large international standard, ISO 9945. This standard will, over time, grow to encompass real-time, security, windowing systems, networking, and so forth. ISO 9945 is, understandably, going to become a very large set of requirements. ISO 9945—standard addresses far too many diverse interests. In addition, even the features of POSIX.1 are too much for small, embedded kernels. These embedded systems have never been able to support even the most basic of UNIX functionality, the fork system call, which duplicates a process context. Yet there is much of POSIX.4 and POSIX.4a that is of crucial interest to applications writers. How will these people be able to determine which systems meet their needs for real-time functionality? How will they be able to avoid being deceived by spurious claims of POSIX conformance, which may be backed up by little more than three or four POSIX calls?

The answer lies in POSIX profiles. A profile can be looked at in two ways. From the system vendor's point of view, it specifies what a particular application area is interested in. From the application writer's point of view, it identifies the facilities that a system must have in order to be useful to the application. A profile is little more than a set of required functionality called out from base standards. For instance, POSIX 1003.13 contains a minimal real-time system profile. This profile includes the threads of 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.

POSIX 1003.13 includes four profiles. At the low end is the aforementioned minimal profile; at the high end is a multi-purpose, "kitchen sink" real-time profile that includes just about all of POSIX.4, POSIX.4a, POSIX.1, and POSIX.2. In between are two medium-sized profiles based on avionics practice (the dedicated-purpose profile) and on embedded systems slightly larger than the minimal (the controller profile).

What is "POSIXness"?

Given that there can be POSIX systems as small as a minimal real-time system or as big as a supercomputer, it is important to have a good handle on exactly what one is getting in a system incorporating POSIX features. Many think of POSIX as being a UNIX standard; it is possible that such people will be tricked into thinking that any system associating itself with the word POSIX has a full complement of UNIX features. This is not so. A UNIX-like system must contain, at a minimum, the full complement of POSIX.1 features (the ability to run multiple processes is crucial, and most minimal real-time kernels will not be able to do this), the full set of POSIX.2 commands (the UNIX command interpreter shell and a significant portion of the usual UNIX commands), and, of course, the ability to write POSIX-conformant applications (a language binding, probably either C or Ada).

USING POSIX NOW

The POSIX.4, POSIX.4a, and POSIX.13 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. POSIX.4 will probably be approved by the IEEE by the end of 1992, with POSIX.4a perhaps a year behind it. POSIX.13 cannot become an official standard until the base standards it relies on (POSIX.4 and POSIX.4a) become finalized. Thus, POSIX.13 will not be signed, sealed, and delivered until some time in 1994.

Meanwhile, it is not too early to begin using the POSIX facilities for real-time. Products such as LynxOS 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. Exceptions to this statement are the message-passing facility, which was pruned back substantially, and binary semaphores, which became counting semaphores. These changes are not as radical as one might think. First, message passing has had many baroque options removed. The basic functionality send a message, receive a message—has not changed; and that functionality is all that the majority of applications will use. As for semaphores, counting semaphores are a generalization of binary semaphores; applications coded to use binary semaphores correctly will be able to switch over to counting semaphores with no change in program logic. Most changes visible to the application developer today will be in the nature of renaming system interface calls, rearranging parameters and re-naming 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.

However, POSIX.13 is a thin, simple document, little more than a set of chapters from POSIX.4 and POSIX.4a; it will be "final" in all but name, long before POSIX.4a is finalized.
Currently, POSIX 1003.4 stands at Draft 13 and is very solid. Few changes are anticipated. POSIX.4a stands at Draft 6 and is rather less solid. POSIX.13 is at Draft 5 at this writing; Draft 6 should be available soon.

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. 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 has real-time POSIX features is not sufficient to guarantee portability of performance. For portability, applications writers must be aware of POSIX profiles, and must require their systems vendors to provide a suitable POSIX profile for their application. Keep in mind that many small real-time kernels are miles away from providing a real UNIX operating system interface; if that is what is required, one must make sure that all of POSIX.1 and POSIX.2 are provided. In terms of performance, only POSIX functionality is required; performance is optional and beyond the scope of the POSIX standards. In our experience, 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, remember that the vast majority of UNIX systems are not capable of supporting real-time applications' performance requirements. Such systems can easily, though, provide the real-time POSIX facilities! Implementing the POSIX interfaces will not make these systems any more capable of real-time performance. Therefore, the applications writer must be careful to demand real-time performance, as well as real-time features.

BIBLIOGRAPHY

LEIBOWITZ91
Michael R. Leibowitz.
The New Generation of RISC Chips
UNIX World
August, 1991, pages 71-76.

LEWINE91
Donald Lewine.
POSIX Programmer's Guide

SVR4
UNIX Software Operation.
UNIX System V Application Binary Interface

MOTORBS
880pen Consortium, Ltd.
880pen Binary Compatibility Standard, Release 1.0

GART90
Mitchell Gart.
Ada and Binary UNIX Standards
Winter USENIX Proceedings

SCHWAN88
Karsten Schwan, Tom Bihari, Bruce W. Weide, and George Taulbee.
High-Performance Operating Systems Primitives
for Robotics and Real-Time Control Systems
ACM Transactions on Computer Systems

AXFORD90
Tom Axford.
Concurrent Programming
Currently, POSIX 1003.4 stands at Draft 13 and is very solid. Few changes are anticipated. POSIX.4a stands at Draft 6 and is rather less solid. POSIX.13 is at Draft 5 at this writing; Draft 6 should be available soon.

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. 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 has real-time POSIX features is not sufficient to guarantee portability of performance.! For portability, applications writers must be aware of POSIX profiles, and must require their systems vendors to provide a suitable POSIX profile for their application. Keep in mind that many small real-time kernels are miles away from providing a real UNIX operating system interface; if that is what is required, one must make sure that all of POSIX.1 and POSIX.2 are provided. In terms of performance, only POSIX functionality is required; performance is optional and beyond the scope of the POSIX standards. In our experience, 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, remember that the vast majority of UNIX systems are not capable of supporting real-time applications’ performance requirements. Such systems can easily, though, provide the real-time POSIX facilities! Implementing the POSIX interfaces will not make these systems any more capable of real-time performance. Therefore, the applications writer must be careful to demand real-time performance, as well as real-time features.

BIBLIOGRAPHY

LEIBOWITZ91
Michael R. Leibowitz.
The New Generation of RISC Chips
UNIX World
August, 1991, pages 71-76.

LEWIN91
Donald Lewine.
POSIX Programmer’s Guide

SVR4
UNIX Software Operation.
UNIX System V Application Binary Interface

MOTORBCS
880open Consortium, Ltd.
880Open Binary Compatibility Standard, Release 1.0

GART90
Mitchell Gart.
Ada and Binary UNIX Standards
Winter USENIX Proceedings

SCHWAN88
Karsten Schwan, Tom Bihari, Bruce W. Weide, and George Taulbee.
High-Performance Operating Systems Primitives
for Robotics and Real-Time Control Systems
ACM Transactions on Computer Systems

AXFORD90
Tom Axford.
Concurrent Programming

BOG91

Bill O. Gallmeister
Reconciling UNIX, Ada, and Real-Time Processing
Dr. Dobb's Journal
June, 1991, pages 57-64.

BAU90

Robert Bauer.
How Real is Real-Time UNIX?
UNIX Review
September, 1990, pages 82-87.

MCJONES87

Paul R. McJones and Garret F. Swart.
Evolving the UNIX System Interface to Support Multithreaded Programs

JONES91

Michael Jones.
Bringing the C Libraries With Us into a Multi-Threaded Future
Winter USENIX Proceedings

BUCK91

A. Lester Buck and Robert F. Coyne, Jr.
An Experimental Implementation of Draft POSIX Asynchronous I/O
Winter USENIX Proceedings

POSIX1

Institute of Electrical and Electronics Engineers.
IEEE Standard Portable Operating System Interface for Computer Environments (POSIX)
1003.1-1990
(Reference Number ISO/IEC 9945-1 : 1990(E))
Institute of Electrical and Electronics Engineers, New York, NY, 1990.

POSIX4

IEEE P1003.4 Working Group.
Available from IEEE 1003.4.

POSIX4A

IEEE P1003.4 Working Group.
Available from IEEE 1003.4.

POSIX13

IEEE P1003.4 Working Group.
Available from IEEE 1003.4.

BOG91
Bill O. Gallmeister
Reconciling UNIX, Ada, and Real-Time Processing
Dr. Dobb's Journal
June, 1991, pages 57-64.

BAU90
Robert Bauer.
How Real is Real-Time UNIX?
UNIX Review
September, 1990, pages 82-87.

MCJONES87
Paul R. McJones and Garret F. Swart.
Evolving the UNIX System Interface to Support Multithreaded Programs

JONES91
Michael Jones.
Bringing the C Libraries With Us into a Multi-Threaded Future
Winter USENIX Proceedings

BUCK91
A. Lester Buck and Robert F. Coyne, Jr.
An Experimental Implementation of Draft POSIX Asynchronous I/O
Winter USENIX Proceedings

POSIX1
Institute of Electrical and Electronics Engineers.
IEEE Standard Portable Operating System Interface for Computer Environments (POSIX)
1003.1-1990
(Reference Number ISO/IEC 9945-1:1990(E))
Institute of Electrical and Electronics Engineers, New York, NY, 1990.

POSIX4
IEEE P1003.4 Working Group.
Available from IEEE 1003.4.

POSIX4A
IEEE P1003.4 Working Group.
Available from IEEE 1003.4.

POSIX13
IEEE P1003.4 Working Group.
Available from IEEE 1003.4.