8. REAL-TIME STANDARDS

8.1. Standard interfaces

Estimates are that some 60 percent of real-systems designers are moving from roll-your-own real-time operating systems to commercial offerings. But talks with designers reveal that off-the-shelf RTOS products still fall short in several significant ways. Designers complain about the lack of a common means of interfacing with kernels, restrictive CPU support, excessive memory requirements and difficulties with multithreading.
Two areas where designers still rely on in-house kernels are low-end 8-, 16- and 32-bit applications with limited memory space, and high-performance 32-bit applications such as networking. According to some estimates, as many as 80 percent of real-time designers writing 8-, 16- and low-end 32-bit applications still build their own kernels in house. And in the high-performance area, one Intel Corp. estimate shows that as many as 70 percent of i960 applications, especially in networking, make use of in-house operating systems.

Off-the-shelf RTOS vendors face a number of challenges. They must find some way of providing OS services to an increasingly diverse population of CPU architectures, both RISC and CISC, as well as increasingly application-specific real-time computer systems. They must also deal with the fact that many real-time applications are not isolated, and need to communicate with other real-time processors.

Moreover, vendors must find ways to solve these problems without further balkanizing a market already full of proprietary, closed solutions. They must respond not only to a demand for more features and capabilities, but also to a need for open systems, or at least a common way of communicating and moving applications across diverse CPU architectures and RTOS implementations.

Perhaps the most prevalent limitation of commercial RTOSes noted by users is that, until relatively recently, there was no common way of interfacing to the kernels. This limits applications even where the development support is rather extensive.

For example, even though [development tool suites] allow you to write and rewrite code for a number of different hardware platforms, it can only be done in the context of a common RTOS. Because of differences in such things as procedural calls and the way interrupts are serviced, there is no way to get from one environment to the other, unless you have the same underlying operating system

While many RTOS vendors have ported their offerings to multiple-arget CPUs, choices are still somewhat confining, Unfortunately, you are often forced to make a choice between the RTOS that is most effective in a particular application and on a particular target, or one which is slightly less effective but has been ported to a number of different CPUs. Occasionally, if you are lucky, you will find an OS that fits both requirements.

In an attempt to make their products more adaptable and portable, many real-time RTOS vendors are rallying behind a common applications-programming-interface standard called Posix. This move is especially compelling because the technical committees responsible for the standard are now in the process of approving a set of extensions specific to the needs of real-time systems.

A derivative of Unix designed mainly for workstations, Posix with its forthcoming real-time extensions is attractive to users. That's because it specifies only the interface, and not how the functions are carried out.

What that means to me is that I still have a wide range of options in terms of picking an RTOS that fits my needs in terms of features, responsiveness and compatibility There is no standard as to the way this Posix compliance is built in, which could lead to unforeseen problems of compatibility.

This concern arises from the fact that in their rush to provide Posix compliance, RTOS vendors are implementing strategies that could conceivably be mutually exclusive. Some companies, such as Venturcom (Cambridge, Mass.) and Modular Computer Systems, licensed the original AT&T Unix code, rewrote the kernel to be preemptible and deterministic, and are not adding features to make it Posix compliant. Others, such as Lynx Real-Time Systems have taken a "clean room" approach and written a kernel that is aimed at compliance with the Unix/Posix interface.

A third approach, taken by companies such as Wind River Systems and JMI Software Systems, is to add a Posix interface to their real-time kernels via library modules that use the Posix interface. In its new 5.1 version of VxWorks, for example, Wind River supports full 1003.1 Posix calls that map to the RTOS interface. It incorporates such real-time extensions as priority scheduling, clock and timer facilities and real-time files. It also incorporates some real-time extensions not included in the draft standards, such as interrupt control, device control and memory management.

This third alternative is attractive, because that gives the option of linking into the customer's Unix-based system if that is what he desires, or using the proprietary modules.

On the other hand, other experienced developers are concerned that the different approaches to implementing Posix could have varying effects on performance.

It is not so much the possible performance degradation, per se, but the differences in performance. There could be some serious discontinuities in applications that run on different Posix implementations.

In spite of improvements such as Posix, designers at the low end of the microcontroller market are still creating in-house real-time kernels and OSes. That's because memory space and addressability rule out the use of what have, until recently, been the memory-hogging requirements of commercial RTOSes.

Memory limitations in the 8- and 16-bit microcontroller world impact the sophistication of the application. Most of the time you are working in RAM space limited to 1 to 3 kbytes and ROM space not very much larger. At that level, it is very hard to separate the application from the operating environment. So more often than not the kernel is the application and vice versa.

In the past, designers have developed the kernel in the machine or assembly code of the CPU on which they were working. But recent improvements in C compilers have resulted in quite impressive improvements in code density that make it possible to write kernel code that fits into relatively tight space.

Meanwhile, companies such as Integrated Systems Inc. with its recently introduced pSOSelect, and Microtec Technology with its VRTXmc, are looking to address the microcontroller segment with space-efficient, scaled-down versions of their RTOSes.

Taking another approach, Wind River Systems has customized a version of its VxWorks RTOS for General Motors for use in memory-constrained microcontroller applications that require no more than 1.5 kbytes for the run-time size of the kernel. With extensions for tools and I/O, memory requirements expand only to 8 kbytes of ROM and 2 kbytes of RAM. This scaled version of VxWorks will be introduced early next year as a standard product.

Ideally, what we would really like is one of those off-the-shelf RTOSes, such as pSOS, VRTX or VxWorks, with not only a standard set of kernel features and services, but all of the tool support that goes with it. It would save a lot of time and money if engineers could work with the same RTOS and the same tools no matter at what level in the real-time-control hierarchy they are working.

At the 32-bit end of the real-time-control spectrum, meanwhile, a number of users continue to rely on operating systems and kernels developed in house. That's because the existing commercial RTOSes are neither sufficiently flexible to allow companies to adapt to changes in the market segments they are targeting, nor do they provide sufficient response time or predictability.

In the network environment, things are changing so fast, with new hardware solutions and networking protocols proposed and tried, that an off-the-shelf operating system with standardized services and features seldom fits very neatly.

And while some new standard RTOSes, such as ISI's pSOS and Microtec's VRTX, use a modularized approach that allows the user to add or subtract features, that there just is not sufficient granularity in such offerings. To address such needs, ISI has proposed the concept of scalable kernels. *Unlike traditional microkernels that work by distributing kernel facilities to a memory space outside the kernel*, a scalable microkernel lets the user balance kernel resources against application requirements and memory constraints.

Such an architecture should also be object-oriented internally, so that the individual, predefined kernel objects could be recombined in various ways to accommodate the needs of the application environment..With any such standard OS or kernel, we are limited to manipulating the the kernel objects defined by the vendor.

Another area where roll-your-own enthusiasts fault commercial RTOSes is in the lack of multithreading, which can improve real-time response. Extensive use is made of multithreading, both for improved performance and for enhanced communications between nodes in a system. Unlike traditional processes such as those used in Unix and many off-the shelf RTOSes, threads are not self-contained, independently schedulable entities that hold less context information about the application environment in which they operate. Because they contain less context information, threads can be started, switched and terminated much faster than processes, On the other hand, one reason they are faster is that they contain less information.

The context information in a process typically contains data relating to registers, address translation tables, paths and I/O descriptors. A thread contains little of this. As a consequence, threads are limited to including information about relatively simple objects, such as a program counter. The trade-off is that this more limited context provides finer granularity.

While a number of off-the-shelf RTOS vendors-including Lynx, Microware, Microtec Research and Wind River-are moving to multithreading, One sign of hope is that among the real-time extensions to Posix now being considered is one that supports multithreading. That may enforce some commonality among the various RTOS vendors.

8.2. POSIX interface

The POSIX specification is a standard developed by the Institute of Electrical and Electronics Engineers (IEEE) to define the interface of the operating system at a high level to provide program compatibility at the source code level and to promote platform independent code. POSIX stands for: Portable Operating System Interface for Computer Environments and is loosely based on the UNIX operating system. The specification is part of the ISO/IEC 9945 and defines the Systems Application Program Interface (API) for the development of "open" or portable code in UNIX-like environments. There are various parts to the complete POSIX specification the most applicable sections are:

- POSIX.1. Operating system interface
- POSIX1a Extensions for real-time
- POSIX.4a Threads Extensions

The difference between POSIX Conformance and Compliance is based in the definitions of the terms. Conformance has a precise and reliable legal definition describing the complete adherence to a set, precise, agreed-upon standard. Compliance, on the other hand has a more vague definition; to yield to ones wish, to agree a requirement. The term "Compliance" is often used by marketing to imply the same set of functionality when in fact this not the case. If a product has not been verified and certified by NIST is should be assumed NOT to conform to the POSIX specifications. For more information about POSIX conformance please consult the article in the February 21, 1994 issue of Embedded Systems by Gurjot Rajpal entitled, "POSIX: Conformance Vs. Compliance".

8.2.1. POSIX.1 (spec. 1003.1)

The POSIX standard has evolved out of the basic pressures from the industry to have some form of compatibility; as historically applications could not be easily moved to different operating systems or hardware architectures. As the result of POSIX conformance, applications can now run on open architectures which provide the following direct advantages for users and developers:

- Vendor independence
- Protection against technological obsolescence
- Interoperability between dissimilar architectures
- Increased availability of off-the-shelf software
- A larger pool of trained programmers and users already familiar with a standard environment.
- Reduced overall costs for project development.

The POSIX.1 specification group is the Systems Application Program Interface (API) which is language independent. The specification provides the definition of the operating system interface at a high level to provide program compatibility at the source code level and to promote platform independent code. The POSIX.1 specification provides the POSIX standardized interface for the following functions:

- File system organization
- System configuration
- Error messages
- Application environment
- Process creation, management, and termination
- Process environment
- Exception conditions
- Timer operations
- File and directory operations
- Record and file locking
- Device specific functions
- C Language specific routines
- User and group database information
- Data interchange formats

8.2.2. POSIX.4 (spec. 1003.4)

The POSIX.4 specification group is the Realtime Extensions to the base POSIX.1 specification. The POSIX.4 specification provides the POSIX standardized interface for the following functions:

- Asynchronous events
- Priority interrupts and scheduling
- Preemptive scheduling
- Memory locking
- High-performance file system (Contiguous Files) (POSIX Realtime Files)
- Realtime timers
- Shared memory
- Semaphores
- Message passing
- Synchronous I/O

An OS must first be POSIX.1 conformant before POSIX.4 conformance can be achieved. The integration of the POSIX.4 specification into the POSIX.1 specification is known as POSIX.1b.

8.2.3 POSIX.4a (spec. 1003.4a)

The POSIX.4a specification group is the Threads Extensions to the POSIX.4 Realtime Extensions specification. The POSIX.4a specification provides the POSIX standardized interface for multiple flows of control in the same POSIX.1 process. A process can be divided into tasks of execution which are independent of each other but work in the same process space.

8.2.4. POSIX.1b (spec. 1003.1b)

The POSIX.1b specification is the application of the POSIX.4 specification to the POSIX.1 specification. The combined specification is referred to as POSIX.1b.