OPEN SYSTEMS FOR TIME-CRITICAL APPLICATIONS IN TELEMETRY

Borko Furht, David Joseph, David Gluch, and John Parker

Modular Computer Systems, Inc., an AEG company
1650 West McNab Road, P. 0. Box 6099
Fort Lauderdale, Florida 33340-6099

ABSTRACT

In this paper we discuss the next generation of open real-time systems for time critical applications in telemetry.

Traditionally, real-time computing has been a realm of proprietary systems, with real-time applications written in assembly language. With the escalating cost of software development and the need for porting real-time applications to state-of-the-art hardware without massive conversion efforts, there is a need for real-time applications to be portable so that they can be moved to newer hardware platforms easily. Therefore, the next generation of real-time systems will be based on open systems incorporating industry standards, which will reduce system cost and time to market, increase availability of software packages, increase ease-of-use, and facilitate system integration.

The open real-time system strategy, presented in this paper, is based on hardware architectures using off-the-shelf microprocessors, Motorola 680X0 and 88X00 families, and the REAL/IX operating system, a fully preemptive real-time UNIX operating system, developed by MODCOMP.

1. INTRODUCTION

The ground-based portion of a modern telemetry system involves a number of key components [1]. At the heart of the system is a high performance real-time data processing, data management, data analysis, and engineering interfaces computer system. One characteristic of current and future versions of these systems is an ever increasing level of computer processing power. This level of processing necessitates the introduction of multiple high performance processing units and high levels of interconnectivity of these processing units [1, 2].

By incorporating industry standards, open systems can meet the demanding requirements of ground-based telemetry data systems by offering the latest in off-the-shelf hardware and software technology plus the ability to network the various processing subsystems into a coherent integrated processing system. In general, open systems reduce system cost and time to market, increase the availability and portability of software packages, increase ease-of-use, and facilitate system integration.

In this article we present a design strategy and specific system designs for the next generation of ground-based real-time telemetry systems based upon open real-time system design principles.

In one of two companion papers in this session [3] the performance measures for these systems are presented. A specific application system installation is presented in the second paper [4).

2. MODEL OF AN OPEN REAL-TIME SYSTEM

A real-time computer system can be defined as a system that performs its functions and responds to external, asynchronous events within a predictable (or deterministic) amount of time. In addition to sufficient and deterministic computational power to meet processing and timing requirements, real-time computer systems must provide efficient interrupt handling capabilities to handle asynchronous events, and high I/O throughput to meet the data handling needs of time-critical applications.

The next generation of real-time systems will use open systems, which incorporate industry standards. Open systems based on standards, provide a number of advantages for users, as illustrated in Table 1.

The ultimate goal of the open system concept is to provide complete software portability. One version of a real-time application should run on various hardware platforms.

The result is that the user is provided with greater variety and greater availability in products; and most importantly a lower cost for both hardware and software. Thus, rather than groups of captive single or limited suppliers, the real-time marketplace is transformed into a broad multiple supplier, competitive pricing environment.

The proposed model of an open real-time computer system, shown in Figure 1, is based on the following principles:

1. The operating system is a standard fully-preemptible real-time operating system, preferably a real-time UNIX™ operating system. It should comply with generally accepted industry standards, such as the IEEE POSIX.

Table 1
Open versus Proprietary Systems

Advantage for Users	Proprietary System	Open System:
Software Portability	Months/Years	Hours/Weeks
Database Conversion	Years	Hours/Days
Programmer Retraining and Availability	Big Issues	Negligible
Flow of Enhancements	Controlled by Computer Manufacturer	Free Market for Major Innovations
Lower Cost	Single or Limited Supplies	Multiple Suppliers (competitave pricing)

2. The system architecture is centralized or distributed, depending on the application. The node architecture in the distributed system, or the architecture of the centralized system, uses a multiprocessor topology based on off-the-shelf microprocessors. The interconnection topology for processors and I/Os is designed to provide extensive I/O, high-speed data processing, and efficient interrupt handling.

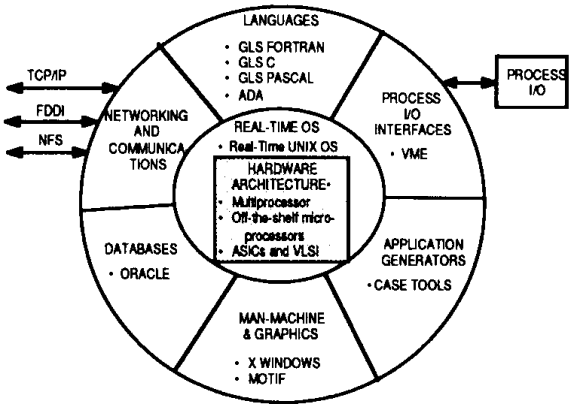

Figure 1. Model of an open system for real-time applications

In addition, ASIC and VLSI technology is used to improve real-time performance by providing architectural supports for operating system scheduling, interrupt handling, fault tolerance, and specific language features.

3. The open system uses a General Language System (GLS) as a standard compiler system. The GLS compiles various source languages into the same intermediate language and therefore provides source code portability on different machines. Currently, the programming languages used in real-time computing are standard languages, such as C, Pascal and Fortran; however, as the complexity of real-time systems increases, there will be an increasing demand for real-time programming languages.

4. The open real-time system provides user-friendly graphics-based man-machine interfaces, which are based on standards, such as the X-Window system and the OSF/Motif™ user environment, which are adapted for real-time use.

5. Open systems support a standard database management system, such as ORACLE™, and the next generation of real-time systems will require distributed real-time databases.

6. Open systems provide connectivity through standards interfaces. For real-time communications, a standard network protocol such as TCP/IP, modified to provide predictable throughput, is used.

   A standard I/O bus such as the VME or Future bus is used for process I/O and peripheral devices. A standard network file server such as NFS is used for transferring files.

7. Standard CASE tools are supported by the system. A set of application generators for various markets is provided.

3. REAL-TIME UNIX OPERATING SYSTEM

An operating system forms a layer of software between the programmer and the base machine and represents the heart of a real-time system.

Requirements for real-time operating systems can be summarized as follows (Table 2).

Table 2
Requirements for Real-Time Operating Systems

- Support for scheduling of real-time processes
- Preemptive scheduling
- Guaranteed interrupt response
- Interprocess communication
- High speed data acquisition
- I/O support
- User control of system resources

1. Support for scheduling of real-time processes

A real-time system should provide support for creation, detection, and scheduling of multiple tasks, each of which monitors and controls some portion of a total application. Typically, a priority-based scheduling of real-time processes will allow users to define priorities for tasks and interrupts to be scheduled. In contrast, in a general-purpose multitasking operating system, the operating system itself determines the order in which tasks execute.

2. Preemptive scheduling

A real-time operating system must ensure that a high-priority task, when ready, preempts a lower priority task. The operating system must be able to recognize the condition (usually through an interrupt) preempt the currently executing task, and perform a fast context switch to allow a higher priority task to execute. A general-purpose operating system, such as the UNIX operating system, will first complete the execution of the task currently running in the kernel unless that task is blocked, and then activate the higher priority task.

3. Guaranteed interrupt response

A real-time system must be able to recognize the occurrence of an event and as quickly as possible take deterministic action based on the event. A real-time operating system must be able to respond on both hardware and software interrupts. The operating system itself should be interruptible and reentrant. There should be minimal overhead in the operating system, especially when context switching to a high-priority real-time process which is ready to execute.

4. Interprocess communications

A real-time operating system must be able to support interprocess communications via reliable and fast facilities, such as semaphores, shared memory and message passing. These facilities are used to synchronize and coordinate task execution, as well as for the use and protection of shared data and shared resources.

5. High speed data acquisition

A real-time system must be able to handle very high burst rates in high speed data acquisition applications. Therefore, a real-time operating system must provide a means to optimize data storage on a disk, moving of the disk heads, moving data from system buffers to user buffers, etc. Some of the features required include the ability to preallocate contiguous disk files, provisions for user control over buffering and so on.

6. I/O support

Real-time applications typically include a number of I/O interfaces. A real-time operating system must be able to provide tools for easy incorporation of custom I/O drivers in the system. For standard devices, the standard I/O library should be available. The operating system must also support asynchronous I/O. Through asynchronous I/O, a task or process can initiate an I/O operation, and then continue execution while the I/O operation is performed concurrently.

7. User control of system resources

A key characteristic of real-time systems is the ability to provide users with specific control of the system resources including the CPU, memory and I/O. Control of the CPU is accomplished by implementing a priority-based scheduling technique. In addition, real-time timers and timer functions can be directly used by applications to schedule events and track elapsed time.

A real-time operating system must also provide memory locking facilities allowing the user to lock a program or part of a program in memory for faster context switching when an interrupt occurs. The user should also be able to control and guarantee memory allocation for buffers and allow the locking and unlocking of devices and files.

3.1 Real-Time UNIX Implementations

The UNIX operating system, developed by AT&T Bell Laboratories, has become a standard operating system gaining rapid acceptance because of its superior flexibility, portability, and large number of support tools to increase programmer productivity. However, the UNIX operating system was originally designed for multitasking and time-sharing, and therefore the standard UNIX operating system does not have an adequate response time nor data throughput capabilities needed to support most real-time applications.

Numerous attempts have been made to adapt the UNIX kernel to provide a real-time environment. Because each implementation is dependent on the type of processor the specific application code used, it is very difficult to compare performance among all of the real-time versions of the UNIX operating system.

One classification method, that we propose, divides real-time UNIX implementations into six categories. These six categories along with the companies taking these approaches, are summarized in Table 3. These six approaches are discussed below.

Table 3
Classification of Real-Time UNIX Implementations

TECHNIQUES	OPERATING SYSTEM & COMPANY
1. Adding extensions to the standard UNIX operating system	• AT&T System V.4
2. Host-target approach	• VxWorks (Wind River Systems • OS/9 (Microwave) • VRTX (Ready Systems)
3. Integrated UNIX and real-time executives (or real-time OS)	• MTOS-UX (IPI) • RTUX (Emerge Systems, Inc.) • CXOS (Computer S/Motorola) • D-NIX (Diab Systems) • real/IX (MODCOMP)
4. Proprietary UNIX operating system	• AIX (IBM) • LynxOS (Lynx Real Time Systems) • Regulus (Alcyon)
5. Preemption points	• RTU (Concurrent/Masscomp) • HP-UX (Hewlett Packard) • VENIX (VenturCom)
6. Fully preemptive kernel	• REAL/IX (MODCOMP) • CX/RT (Harris)

1. Adding Extensions to the Standard UNIX Operating System

In this implementation approach to a real-time UNIX operating system, extensions are added to the core UNIX operating system. The real-time extensions can be implemented in the kernel or outside of the kernel. This is the approach taken by system-level vendors, such as AT&T in its System V.4 UNIX release.

Some examples of adding extensions to the UNIX Operating system include priority-based scheduling in a task scheduler, and real-time timers and priority disk scheduling.

Although this approach provides real-time functionality, it has its drawbacks. The main drawback is the absence of full preemption in the kernel mode. When an application task makes a system service call, and the task goes into kernel mode, the system has to complete that service call before a higher priority task can get the CPU.

2. Host/Target Approach

The host/target approach to using the UNIX operating system in a real-time environment is to develop an application on a UNIX host and then download it to a proprietary real-time kernel or to another operating system running on a target system.

The operating systems that use the host/target approach are given in Table 3. For example, in the case of VxWorks operating system, developers use the UNIX host for development, that includes editing, compiling, linking, and storing real-time applications. Then the program is tested, debugged and executed on the target machine running VxWorks. The host and target communicate via sockets and TCP/IP protocol.

The VRTX operating system also allows communications between the target processor running the VRTX real-time kernel and the host computer that runs the UNIX operating system. For communications, a shared-memory implementation is used.

The host/target approach combines the advantages of the UNIX operating system with those of a proprietary real-time kernel. The response time of a reduced functionality proprietary kernel (under 10 microseconds) is still faster than the response times of real-time UNIX implementations (from 70 microseconds to a few milliseconds). However, this approach requires two operating systems and the porting of application software to other platforms is more difficult.

3. Integrated UNIX and Real-time Executive/OS

This approach provides a UNIX interface to a proprietary real-time kernel or a proprietary real-time operating system. Both UNIX and the proprietary kernel/OS run on the same machine. The MTOS real-time kernel has been developed, for example, which can log onto a UNIX-based system. The real/IX™ operating system integrates the AT&T System V with MAX 32 real-time operating System. The user can utilize the development tools in the UNIX environment to develop the real-time application, and then run that application in the proprietary operating system environment.

This approach provides the fast real-time response of the proprietary real-time systems, however it requires two operating systems, and the porting of applications is more complicated.

4. Proprietary UNIX

This approach to a real-time UNIX operating system consists of developing a proprietary real-time kernel from the ground up while maintaining the standard UNIX interfaces. For example, Lynx OS, AIX, and Regulus operating systems use this approach. Their internal implementations are proprietary, however the interfaces are fully compatible with a standard UNIX operating system such as the AT&T System V operating system. These interfaces are specified with standards such as SVID and IEEE POSIX.

This approach provides relatively high-performance, however porting real-time applications to or from another real-time implementation of UNIX requires rewriting code. Standard UNIX applications will run under these operating systems, but code that uses the real-time extensions will need to be rewritten.

5. Preemption points

One of the most critical requirements for a true real-time implementation of a UNIX operating system is kernel preemption. Most of the non-real time operating systems implement a round-robin scheduler, in which a real-time task is added to run queue to await its next slice.

In the standard UNIX operating system, a process executing in the kernel space is not preemptible. A system call, even one from a low-priority user process, continues executing until it blocks or runs to completion. This can take as long as several seconds.

Another approach to implementing a real-time UNIX operating system is to insert preemption points, or windows into the operating system kernel. Preemption points have been built into the kernel, so that system calls do not have to block or run to completion before giving up control. This can reduce the delay before the higher-priority process can begin or resume execution. However, as an impact of preemption points, there is still a preemption delay which may be as high as several milliseconds. This delay corresponds to the longest period of time between preemption points. The preemption points approach is taken by RTU, HP-UX, and VENIX operating systems. The RTU operating system includes approximately 100 preemption points and 10 preemptible regions. At the preemption points, the operating system performs a quick check to see if a real-time process is ready to run. Preemption regions are sections of the kernel in which the scheduler is always enabled.

The preemption points approach provides some degree of determinism and fast response time, but it is not a fully preemptive system and the drawback to this approach is similar to the drawback of the adding extensions approach previously described.

6. Fully preemptive kernel

A fully preemptive real-time UNIX operating system allows full preemption anywhere in either the user or the kernel level. The preemptible kernel can be built by incorporating synchronization mechanisms, such as semaphores and spin locks, to protect all global data structures. By implementing a fine granularity of semaphores, the preemption delay can be reduced to approximately 100 microseconds.

A fully preemptive kernel provides the system with the ability to respond immediately to interrupts, to break out of the kernel mode, and to execute a high-priority real-time task. This approach is implemented in the REAL/IX and CX/RT operating systems. The VENIX and RTU versions of fully preemptive UNIX kernels are under development.

3.2 Case Study: The REAL/IX Operating System

In this section the features of the REAL/IX operating system, a fully preemptible real-time UNIX operating system, are discussed [6].

A major design feature that provides deterministic behavior and high performance interrupt handling within the REAL/IX system is the fully preemptive nature of the kernel. In standard implementations of the UNIX operating system, a process executing a system call (i.e., operating within the kernel system space) cannot be preempted. Regardless of the priority of the calling process, system calls execute until they block or run to completion. In contrast, within the REAL/IX operating environment, lower priority processes can be interrupted at any time and a higher priority process can begin executing.

The preemptive kernel has been implemented through the use of semaphores that protect global data structures. Through the fine granularity of the semaphore implementations the preemption delay has been reduced to approximately 100 microseconds on the MODCOMP Tri-D 9730 system. The 9730 is based upon a 25 MHz Motorola 68030 with 8 to 32 Megabytes of memory.

Priority Scheduling. Scheduling within the REAL/IX operating system is divided into two scheduling strategies: Time-Sharing and Fixed-Priority Scheduling. As shown in Figure 2 the run queue consists of 256 process priority “slots” that are grouped into the two scheduling strategies.

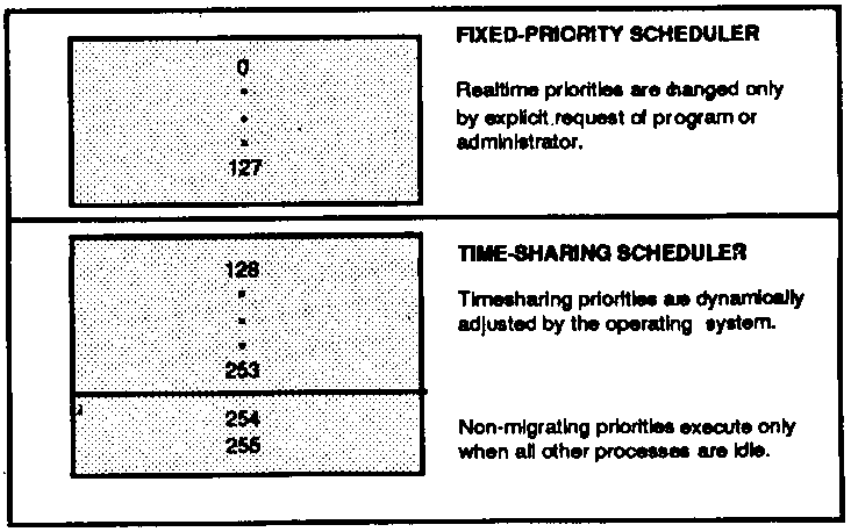

Figure 2. REAL/IX Process Priorities

Each executing process has a priority that determines its position on the run queue. Processes executing at priorities 0 through 127 (real-time priorities) use a process priority scheduler implemented internally. Processes executing at priorities 128 through 255 (time-slice priorities) use a time-sharing scheduler.

Processes are scheduled according to the following rules:

- A process runs only when no other process at a higher priority is runnable.
- Once a process with a real-time priority (0 through 127) has control of a CPU, it retains possession of that CPU until it is preempted by a process running at a higher priority; or relinquishes the CPU by making a call that causes a context switch; or blocks to await some event, such as I/O completion; or its time slice expires.
- A running process can be preempted at any time if a process at a higher priority becomes runnable.

The process table for each executing process includes scheduling parameters used by the kernel to determine the order of execution of processes. These parameters are determined differently for time-sharing and fixed-priority scheduled processes.

Interrupt Handling. Often real-time processes are waiting for some real-world event to occur, manifested as an interrupt within the system, in order to be activated. When the interrupt occurs, a currently executing lower-priority process must quickly be switched out and the real-time process switched in.

Process dispatch latency time can be defined as the interval between the time that the system receives an interrupt request and the beginning of execution of the application task associated with the interrupt. The measures most frequently used in benchmarks, interrupt latency and context switching time, are important real-time parameters. However, the total process dispatch latency time is composed of many components, as shown in Figure 3. The REAL/IX system has a total process dispatch latency time of 67-260 microseconds on the 9730 computer.

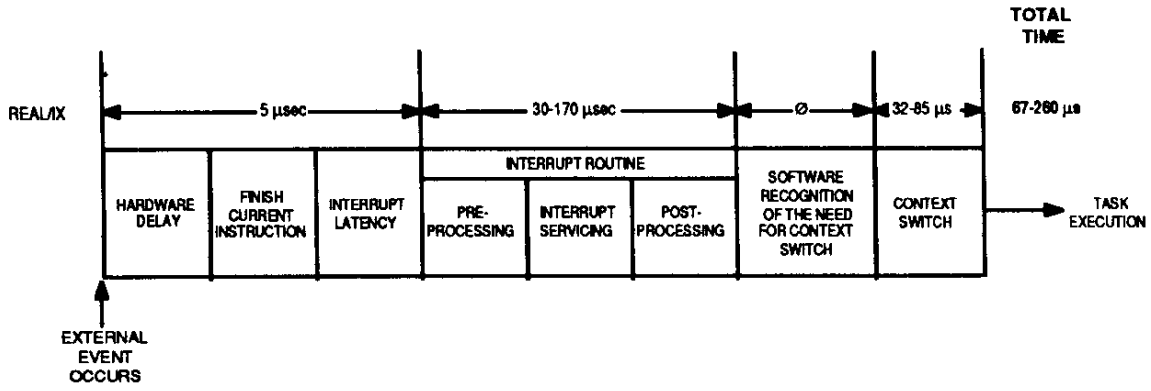

Figure 3. Process dispatch latency time

4. OPEN REAL-TIME ARCHITECTURE

The open real-time system architecture, whether centralized or distributed, uses a multiprocessor topology based on off-the-shelf microprocessors. The interconnection topology provides extensive I/O processing, high-speed data processing, and efficient interrupt handling. The system architecture includes the following features [5].

- architectural support for real-time operating systems,
- architectural support for scheduling algorithms,
- architectural support for real-time languages,
- fault-tolerance and architectural support for error handling, and
- fast, reliable, and time-constrained communications.

In this section we present three case studies of open real-time architectures that are based on the following topologies:

- tightly coupled multiprocessor with common memory,
- host/target multiprocessor with distributed memory, and
- high-availability dual processor architecture.

One of the key features of these open systems is the portability of software across all of the architectures. Software code which executes on one architecture will also run on any of the others.

4.1 Tightly Coupled Multiprocessor with Common Memory

In a tightly coupled multiprocessor system with common memory, all CPUs share global memory, and there is a single copy of the operating system and data structures. In existing tightly coupled multiprocessor architectures, the global memory bus represents a system bottleneck, and therefore such a system cannot be directly used in a real-time environment [7]. However, by modifying the architecture and adding real-time support components, it is possible to adapt the tightly coupled multiprocessor with common memory for real-time applications.

An example of the real-time tightly coupled multiprocessor with common memory is MODCOMP's N4 system, described next [8, 9].

The system, shown in Figure 4, consists of 1 to 10 off-the-shelf microprocessors (in the case of the N4 these processors are 33 MHz MC68030), which share global memory through the high performance Quad Wonder Bus (QWB). They can be configured as processing units (CPUs) or I/O units. A single copy of the REAL/IX multiprocessor real-time operating system resides in global memory and provides total operational control over all processing within the system. In addition, a System Support Processor (SSP) provides diagnostic, maintenance and related system functions. In addition to the QWB, there is an Event Bus (EB). The Event Bus allows the system to handle a large number of interrupts by load balancing real-time interrupts.

Besides the QWB and Event bus, which improve real-time performance of the tightly-coupled multiprocessor system, several other architectural features are incorporated in the system in order to support the real-time operating system. All of these features are described in this section.

The Quad Wonder Bus is the memory bus which connects all CPUs, I/O modules, and the SSP with the global memory. it consists of four independent synchronous buses with non-multiplexed address and data bits. The QWB has a maximum write transfer rate of 160 MB/sec, and a maximum read transfer rate of 245 MB/sec, which gives an average transfer rate of approximately 200 MB/sec. The addressing scheme of the QWB allows sequential access by non-memory agents to be evenly distributed over the four independent buses.

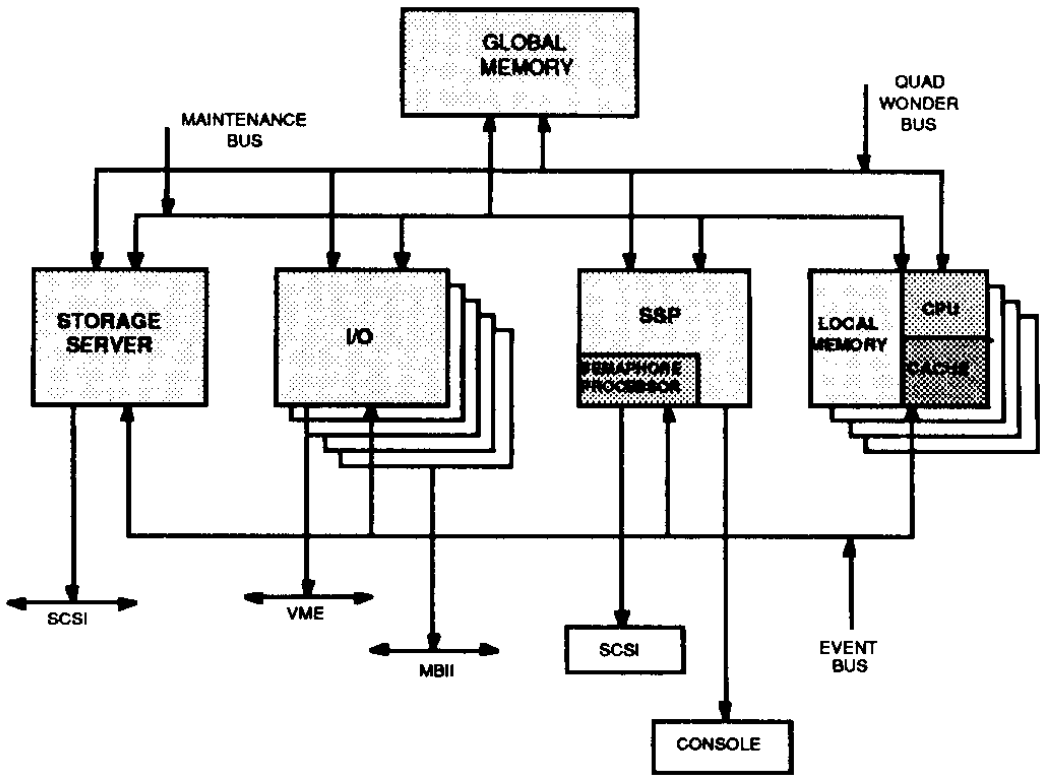

Figure 4. Tightly coupled symmetrical multiprocessor with common memory

The Event Bus is a synchronous multifunction bus. The data bus portion of the bus provides a venicle on which interrupts, priority resolution, and interrupt tagging information are passed. The control portion of the bus defines at any given time the Event Bus operating mode. The Event Bus is the vehicle through which the interrupt requesting agents and interrupt servicing agent resolve all source and destination routing requests. The Event Bus consists of 32 data lines, 4 interrupt identification lines, 3 interrupt cycle function lines and a synchronization clock. The data lines are multiplexed to provide 16 levels of prioritized interrupts and 16 interrupt enable lines.

Each CPU unit consists of: (a) a high-performance 32-bit single chip CPU, which has an on-chip bus interface unit, instruction decode and execution units, and memory management unit, (b) floating point coprocessor, (c) local memory, (d) cache controller and cache memory, (e) local interrupt controller unit, (f) memory bus controller, and (g) semaphore assist logic.

Local memory is used for storing Operating System (OS) instructions, non-user interrupt service routines, and private non-shared data for fast access. Local access of the OS instructions and data increases bus bandwidth to other bus agents by avoiding access of the Memory Bus.

The cache memory, consisting of 128 Kbytes, improves the performance of the global memory (DRAM) by providing the CPU with a small amount of high speed SRAM. The cache contains a copy of frequently accessed global memory locations, thus providing direct access to high performance SRAM based memory.

The Semaphore Processor (SP) is an extension of the CPU/Memory environment which incorporates software semaphore algorithms in a hardware state machine that is connected to the memory Bus via the controllers. The Semaphore Processor consists of a hardware sequencer, 1 Mbyte RAM, and the QWB interface. Software semaphore algorithms are encoded in memory read addresses that are intercepted and serviced by the SP. Within the addresses received, two fields are used to specify the particular type of the algorithm and the semaphore memory address to be operating upon. In this approach, the SP appears to the other components of the system as a memory agent. The QWB protocol is adhered to by SP, minimizing the effort to invoke a semaphore operation. Bus locking is not required due to the SP operating upon the semaphore data on a first-come, first-serve basis.

The performance of the N4 system with 10 processors is about 85 MIPS; it can process 500,000 interrupts per second, and its I/O throughput is 100 Mbytes per second. Simulation and modeling results show that CPU speed is almost a linear function of the number of processors, as illustrated in Figure 5.

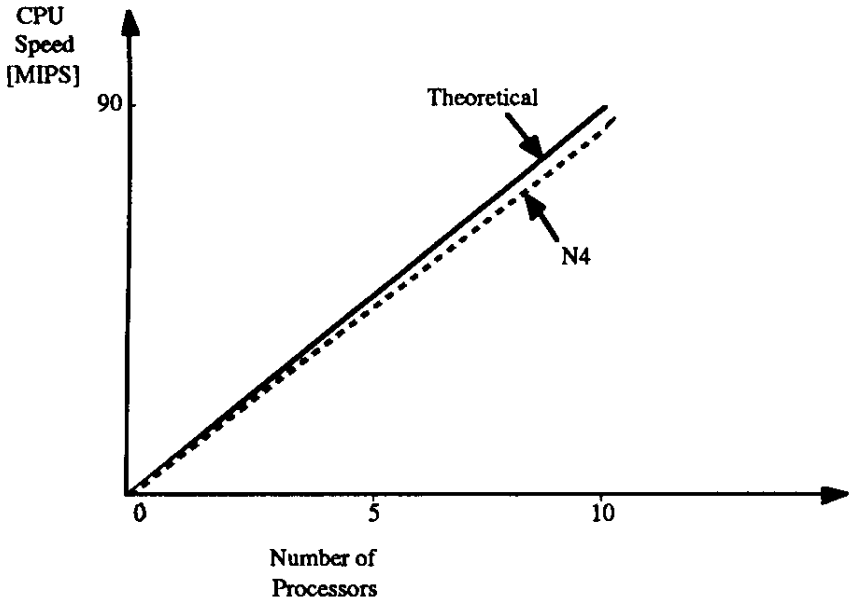

Figure 5. Performance of the N4: CPU speed as a function of number of processors

The N4 system is currently using MC68030 processors, however, future machines can easily be upgraded to MC68040, RISC 88000, and other new processors.

4.2 Host/Target Multiprocessor with Distributed Memory

The host/target multiprocessor with distributed memory addresses a class of real-time problems that operate in a synchronized environment with rapid response to external events, such as real-time simulators and trainer systems.

An example of such a system is MODCOMP's N2+ system, shown in Figure 6, which consists of a system host, up to 8 high performance target computer nodes, and various real-time I/O units.

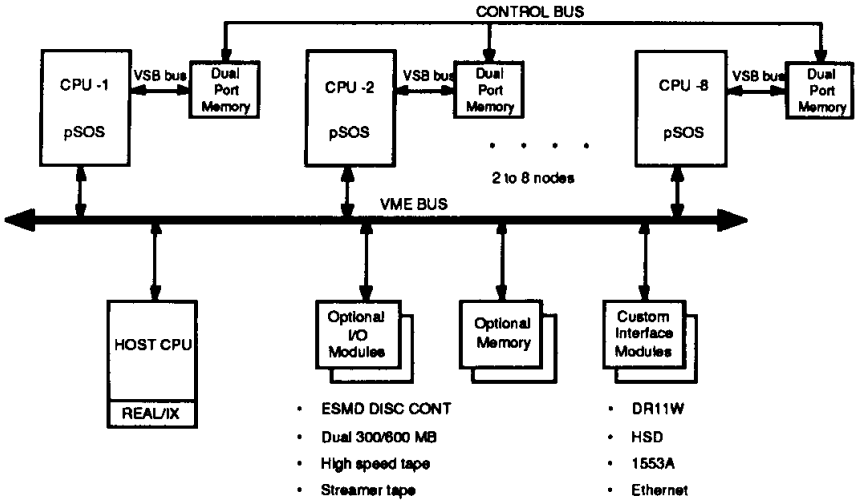

Figure 6. Host/target multiprocessor system with distributed memory

The unique needs of trainers and simulators are addressed through the implementation of a mirror memory/distributed database supported by broadcasting writes and interrupts in the tightly coupled host/target architecture.

The majority of the N2+ hardware system components are commercially available units with the exception of the distributed memory system. The distributed memory system, based on the concept of multiple slaves on the VME bus, allows the broadcast of data and interrupts over the VME bus.

The host and target CPUs are MC68040-based single board computers, while the I/O support is provided by high-speed parallel I/O controllers and various specialized controllers required in trainers and simulators (HSD, 1553B, DR11W, A/D, D/A, etc.).

The computational power of the N2+ system with the host CPU and 8 nodes is 9 X 20 = 180 MIPS.

The N2+ software environment allows a user to develop, debug, test and deploy a distributed real-time application. The system software consists of a system host environment and a target computer node environment. The system host runs under REAL/IX, a real-time UNIX operating system, while the target computer nodes operate under pSOS, a real-time executive.

4.3 High-Availability Dual Processor Architecture

Real-time applications require functionally correct, timely, and reliable computation. To provide enhanced reliability and availability, flexible fault tolerance capabilities should be incorporated into the design of the hardware and software of real-time systems. This flexibility enables the system to adapt to the specific level of reliability and availability required by an application without design modifications and results in a comprehensive general purpose high performance real-time system.

Fault tolerant real-time computer systems can be grouped into three qualitative classes [10, 11]:

- Graceful Degradation Systems
- High Availability Systems
- Continuous Performance Systems

Graceful Degradation Systems do not rely on explicit redundancy but rather utilize the fact that, as part of the system's basic design, there are multiple subsystems for each of the operationally critical functions within the system. Thus, by combining basic fault handling mechanisms with this implicit hardware redundancy, graceful degradation systems can recover to a reduced capability configuration following the occurrence of a fault. These recovery and reconfiguration processes are generally accomplished as off-line functions.

High Availability Systems generally involve high levels of data integrity protection, a low probability of loss of function or of system malfunction, and rapid system recovery. Basically, high availability applications require systems that include at least some capability to continue operation in the event of a failure. These systems include fault detection, fault isolation, and fault response mechanisms, and occasionally on-line repair capabilities; and their complexity can range from simple dual redundancy in a critical subsystem (e.g., dual disk drives) through to total system redundancy.

Continuous Performance Systems are implemented where the cost of even a short interruption in operation results in costly or catastrophic consequences. These systems generally involve very high levels of data integrity protection and very low probabilities of loss of function or system malfunction. Generally, extensive fault masking and on-line repair and recovery techniques are employed in these systems.

A high-availability dual processor architecture that meets the requirements of the first two fault tolerance classes (graceful degradation and high availability) is shown in Figure 7.

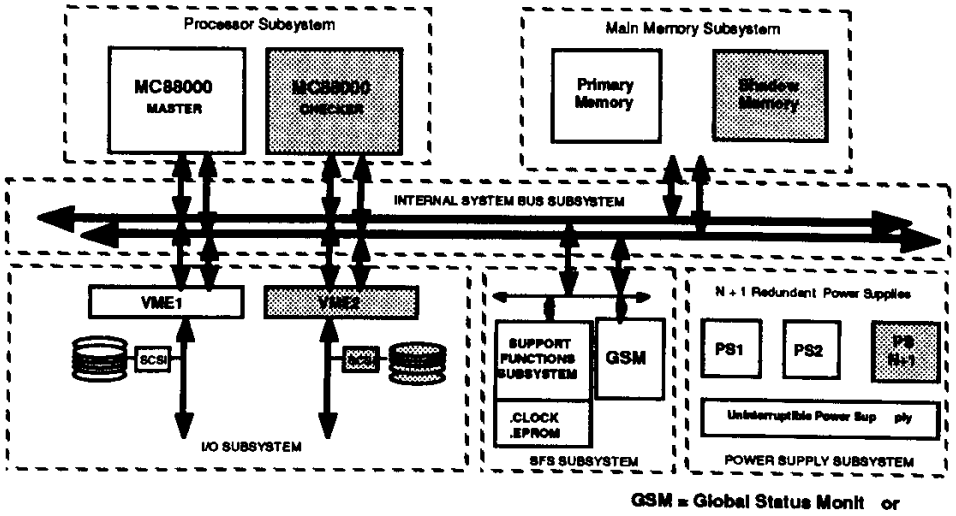

Figure 7. The architecture of a high availability dual processor system

The system, referred to as MODCOMP's C3/FT, includes fail-operational capabilities for processor, memory, internal bus (graceful degradation), I/O, and power source subsystem failures. An uninterruptible power supply is integrated into the overall fault resilience of the system and the fault tolerance functions are closely integrated with diagnostic software to facilitate system maintenance and repair.

The C3/FT architecture consists of six major subsystems: (1) Processor subsystem, (2) Main memory subsystem, (3) Input/output subsystem, (4) Power supply subsystem with Uninterruptible Power Supply (UPS), (5) Internal system bus subsystem, and (6) Support function subsystem and global status monitor.

The processor, main memory, internal system bus, and input/output subsystems include two identical functional units. The power supply subsystem includes multiple functional units. With the exception of the internal system buses, each of the major dual or multiple functional unit subsystems can be configured by the user in either the normal (non-redundant) or in the redundant configuration. The dual internal system buses (M_buses) are employed for high performance during normal operations and are automatically reconfigured to single bus operation following a failure.

The flexibility to configure the processing subsystems defines two primary operational modes for the C3/FT computer system:

(1) Dual Processing (DP) mode and,
(2) Redundant Processing (RP) mode.

In the Dual Processing mode the two processors and associated CMMUs are working individually and thus provide dual processor performance. In the Redundant Processing mode the processors are tightly (clock) synchronized and both are executing identical instructions. In this mode, one processor is checking the operation of the other in a Master/Checker relationship.

The system configured in dual processing mode provides 28 MIPS of computational speed, 600,000 interrupts per second, and 40 Mbytes I/O throughput.

5. CONCLUDING REMARKS

In this paper, the concept of an open real-time system is introduced and several practical system designs, based on this concept, are presented.

The special problems that we emphasized in the paper are related to a standard real-time operating system, such as a real-time UNIX operating system, and a standard open real-time system architecture, based upon off-the-shelf microprocessor. The topics that were just briefly touched upon, but that deserve more attention, are related to specification and verification of real-time systems, real-time programming languages and design methodology, real-time scheduling algorithms, distributed real-time databases, real-time communications, and real-time artificial intelligence.

According to Stankovic [5], many real-time systems of tomorrow will be large and complex and will function in distributed and dynamic environments. For such systems, a theory of large-scale real-time systems should be developed. Such a theory would include the current methodologies of operating systems theory, database theory, system design theory, scheduling theory, and others integrated with new theories and technologies for real-time. For future reading, we suggest the IEEE Tutorial on Hard Real-Time Systems [12].

ACKNOWLEDGMENTS

This article incorporates the ideas of many individuals from the MODCOMP R&D Center. Those who contributed significantly to the ideas presented in this paper, and who were involved in converting these ideas into designs include William Earnshaw, Dan

Grostick, Tom Kapish, Sunil Khandwala, Dave Klein, Meg McRoberts, Hagai Ohel, Bud Parrish, Guy Rabbat and Bill Routt. The authors would also like to thank Ms. Terri Stettner for her invaluable contribution to the production of this article.

REFERENCES

1. Rauch, W. D., "Telemetry Systems of the Future", *Proc. of the International Telemetry Conference*, Inst. Soc. of Amer., 1989, pp. 63-73.

2. Lynch III, T. J., Fortmann, T. E., Briscoe, H., and Fidell, S., "Multiprocessor-Based Data Acquisition and Analysis," *Proc. of the International Telemetry Conference*, Inst. Soc. of Amer., 1989, pp. 51-62.

3. B. Furht, D. Gluch, and D. Joseph, "Performance Measurements of Real-Time Computer Systems", *Proc. of the 1990 International Telemetry Conference*, Las Vegas, Nevada, October 1990.

4. R. Kibler, B. Rodgers, R. Beers, and D. Joseph, "REAL/IX2: The Heart of a Very Fast Real-Time Test Range System," *Proc. of the 1990 International Telemetry Conference*, Las Vegas, Nevada, October 1990.

5. J. A. Stankovic, "Misconceptions About Real-Time Computing," *Computer*, Vol. 21, No. 10, October 1988, pp. 10-19.

6. *REAL/IX Concepts and Characteristics Manual*, MODCOMP, Ft. Lauderdale, 1989.

7. T. N. Mudge, J. P. Hayes, G. D. Buzzard, and D. C. Winsor, "Analysis of Multiple-bus Interconnection Networks", *Journal on Parallel and Distributed Computing*, February 1986, pp. 328-343.

8. B. Furht, G. Rabbat, J. Parker, R. Kibler, W. E. Earnshaw, P. C. Ripy, and H. Ohel, "Tightly Coupled Multiprocessor Structure for Real-Time Applications", Patent No. 89112824.1, 1989.

9. B. Furht, G. Rabbat, R. Kibler, J. Parker, and D. Gluch, "The Design of Tri-D™ Real-Time Computer Systems," *Proc. of the EUROMICRO Workshop on Real-Time*, Como, Italy, June 14-16,1989, pp. 84-92.

10. D. A. Rennels, "Fault-Tolerant Computing: Concepts and Examples", *IEEE Trans. on Computers*, Vol. 33, No. 12, December 1987, pp. 1116-1129.

11. D. P. Gluch and B. Furht, “Fault Tolerance Strategies for High Performance Real-Time Computers,” *International Journal of Mini & Microcomputers*., Vol. 11, No. 2, May 1989, pp. 24-30.

12. J. A. Stankovic, and K. Ramamritham, “Tutorial on Hard Real-Time Systems”, *IEEE Computer Society*, 1988.

NOMENCLATURE

ASIC	Application Specific Integrated Circuits
C3/FT	MODCOMP’s High Availability Dual Processor System
EB	Event Bus
GLS	General Language System
N2+	MODCOMP’s Host/Target Multiprocessor
N4	MODCOMP’s Tightly Coupled Multiprocessor
NFS	Network File Server
QWB,	Quad Wonder Bus
SP	Semaphore Processor
TCP/IP	Transmission Control Protocol/Internet Protocol
VLSI	Very Large Scale Integration
VME	Versa Module Europa