RTEID: THE QUEST FOR REAL TIME STANDARDS

Richard Vanderlin
Paul Raynoha
Brian Hansche
Luke C. Dion

MOTOROLA Microcomputer Division
Tempe, Arizona

ABSTRACT

Software and hardware standards have become important building blocks in the computer industry. This paper will discuss a proposed real time executive standard, its major features, and its relationship to other industry standards.

INTRODUCTION

This paper will present a viewpoint for the adoption of a standard software interface for the development of real time applications. This proposed standard is called the Real Time Executive Interface Definition (*RTEID*). The *RTEID* plays a central role in Motorola's VMEexec real time software development. Major functions of the proposed *RTEID* will be described. We will address the current state of real time executive software, the need for standardization, and the advantages of integrated UNIX/real time development and runtime environments.

SOFTWARE STANDARDS

Promoting standards for software has met with various degrees of success. The standardization effort of the Microprocessor Operating Systems Interfaces (MOSI) has met with only limited success. UNIX has evolved as a popular operating system and several attempts are currently underway to develop a standard interface definition for portability of UNIX applications. The Portable Operating System Interface Specification (POSIX) is an ongoing activity directed to the standardization of UNIX interfaces. The C language is becoming a de facto standard for developing applications for real time systems. Very little, however, has been done to address the needs of the real time community in terms of standards. The lack of standards is due, in part, to the emerging nature of real time systems and applications. The development of real time executives is a relatively new software industry, with most of the popular executives having been introduced within the last six years. With more companies in the business of industrial automation opting for *off the shelf* software rather than *in house* design, it's now appropriate to standardize real time system interfaces.

With this goal in mind, and in an effort to provide the best software solution to our VME board and system customers, Motorola has proposed a real time executive interface standard. The Real Time Executive Interface Definition (*RTEID*) is the result of that effort. The *RTEID* specification was developed by Motorola with technical input from Software Components Group, Inc. (SCG). SCG is the manufacturer of *pSOS*, a widely used real time executive.

While Motorola will actively promote the adoption of the *RTEID*, the standardization process is open to all interested parties.

The advantages of using standards, whether they are formally defined or de facto, are well understood in the industry. Figure 1 illustrates the use of industry standards for real time applications.

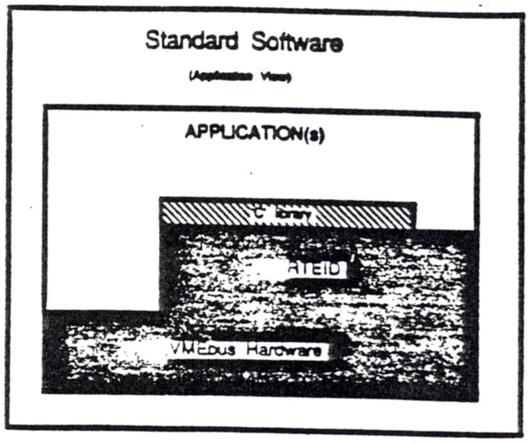

Figure 1. Standard Software Applications

The *VMEbus Hardware* represents the formally approved standard IEEE 1014-87. The *RTEID* will be promoted as a standard, with the goal of having many conforming real time executives available.

Real time application projects which are written in a portable programming language, like C, and which are written to an *RTEID* interface, will have the advantage of portability, hardware flexibility, and improved programmer productivity. For example, rather than designing real time software to work on a predesigned, limited hardware base, real time software can be written and then fitted to the most cost effective hardware base needed for that application. What may be even more important is the insurance against obsolescence that one obtains by adhering to standards. Long term usefulness of real time software will be virtually guaranteed through the use of software standards, hardware standards, and a upwardly compatible family of processors like the Motorola M68000 microprocessor family.

MARKET SURVEY

The initial work on the *RTEID* specification came about as a result of Motorola's effort to develop a new real time executive with new features such as embedded multiprocessing, MMU support for its new integrated microprocessors (such as the M68030 and beyond), and advanced software debug capabilities. While researching existing real time executives, we realized there was a large degree of similarity in functionality among most executives on the market. Consequently, we now believe there is a certain amount of de facto standardization of real time executives already occurring in the marketplace. The similarities of the popular executives help identify basic needs or requirements of any good real time executive. Memory buffer management, task control, message passing, simple interrupt control, and some form of signal or semaphore operations are the most common features. Table 1 contains a side-by-side comparison of major features of five popular real time executives. This table has been reprinted (in part) with permission from the May/June 1987 issues of VMEbus System magazine. (Since that time, new features may have been added to these products by their developers.)

When initially proposing the *RTEID*, Motorola met with considerable opposition from some major VMEbus vendors and real time executive developers. However, several companies (including both real time software producers and consumers) have since become very interested in the *RTEID*. The current draft Version 2.1, has been presented to the VME International Trade Association (VITA) Real Time Software Technical Sub-Committee.

Since the initial draft of the *RTEID* specification was published, several other companies worldwide have expressed an interest in developing *RTEID* compliant executives or compatible products. The time for real time software standards, to the benefit of the entire industry, is quickly approaching.

THE RTEID

The *RTEID* specification defines several functions, also known as directives or services, and groups them into a set of resource managers. Except for the debug manager, which is defined as an optional extension to the standard, all managers are required of any *RTEID* compliant executive. Table 2 shows a complete list of *RTEID* directives and associated arguments.

Resource managers for tasks, queues, events, semaphores, asynchronous signals, time management, memory regions and partitions, and memory management unit support are defined in the *RTEID*.

Task Manager

The *task manager* provides those directives that allow control of a task or collection of tasks. A *task* is a sequence of closely related computations that can execute concurrently with other computational sequences within a multitasking environment. Another name commonly used for *task* is *process*. A task typically accepts one or more inputs, performs

TABLE 1. Silicon Software Components

	pSOS	MTOS	VRTX	PDOS	C-EXEC
KERNEL FEATURES:					
SIZE	5.5K	6.11K	6K	6K	5-7K
ROMable	Yes	Yes	Yes	Yes	Yes
WRITTEN IN	ASSY	C+ASSY	ASSY	ASSY	C
TASK SCHEDULING:					
Prioritised	Yes	Yes	Yes	Yes	Yes
Round-Robin	Yes	Yes	Optional	Yes	Yes
Pre-emptive	Optional	Yes	Yes	Yes	Optional
Dynamic Priorities	Yes	Yes	Yes	Yes	No
Allows Spawning of Tasks	Yes	Yes	Yes	Yes	No
Code Sharing Between Tasks	Yes	Yes	Yes	Yes	Yes
Inter-Task Message Passing	Yes	Yes	Yes	Yes	Yes
FILE SYSTEM SUPPORT:					
Contiguous Files	Yes	Yes	Yes	Yes	Yes
Non-Contiguous Files	Yes	Yes	Yes	Yes	Yes
Record Locking	Yes	Yes	Yes	Yes	Yes
Randon File Access	Yes	Yes	Yes	Yes	Yes
Sequential File Access	Yes	Yes	Yes	Yes	Yes
Time Stamping of Files	Yes	Yes	Yes	Yes	Yes
Provides Disk Caching	Yes	Yes	Yes	Yes	Yes
Redirection of I/O	Yes	Yes	Yes	Yes	Yes
DEVELOPMENT SUPPORT:					
Resident Debugger	Optional	Yes	Optional	Yes	No
Upload/Download Utility	Yes?	No	Download Only	Yes	No
Performance Profiling	Yes	No	Yes	Yes	No
High-Level Language Support					
Basic	No		No	Interpreted	No
C	Lib	Lib	Lib+Comp	Lib+Comp	Lib
Fortran	No		No	Comp	No
Pascal	Lib		Lib	Comp	No
ADA	No		No	No	No
Can Read/Write IBM/PC Diskette	Yes	Yes	Yes	No	Yes
MULTIPLE PROCESSOR SUPPORT:					
Message Passing (Exec-Exec)					
Tightly Coupled	Yes	Yes	Yes	No	No
Loosely Coupled	No	No	Yes	Yes	Yes
Inter-CPU System Calls	Yes	Yes	Yes	No	No
Special Debugging Tools	Yes	No	No	No	Yes
Message Passing (Exec-Unix)					
Tightly Coupled	Yes	No	No	No	No
Loosely Coupled	No	Yes	No	No	Yes

some processing based on the input, and produces one or more outputs.

A task is created using the *t_create* directive. Once a task is created, other tasks can refer to it and act on its behalf to allocate resources for it. A task is started with the *t_start* directive. Once a task has been started, it can execute its program and vie with other tasks for processor time according to its relative priority.

A task may be deleted with the *t_delete* directive. All knowledge of the task is removed from the system, and other tasks referring to it will be returned an error.

All tasks have a task identifier (*tid*). The *tid* is assigned to the task at task creation time, and must be used in subsequent calls to the executive to identify that task. The *t_ident* directive may be used to obtain the *tid* of another task when that task's name is known.

All tasks have a priority. A task's priority is a measure of the task's importance relative to all other tasks within the system. The task's priority indicates

TABLE 2. RTEID Directives

Name	Input Parameters					
TASKS:						
t_create	name	superstk	userstk	priority	flags	&tid
t_ident	name	node	&tid			
t_start	tid	saddr	mode	argp		
t_restart	tid	argp				
t_delete	tid					
t_suspend	tid					
t_resume	tid					
t_setpri	tid	priority	&ppriority			
t_mode	mode	mask	&pmode			
t_getreg	tid	regnum	&regval			
t_setreg	tid	regnum	regval			
MESSAGES:						
q_create	name	count	flags	&qid		
q_ident	name	node	&qid			
q_delete	qid					
q_send	qid	buffer				
q_urgent	qid	buffer				
q_broadcast	qid	buffer	&count			
q_receive	qid	buffer	flags	timeout		
EVENTS:						
ev_send	tid	event				
ev_receive	eventin	flags	timeout	&eventout		
SIGNALS:						
as_catch	asraddr	mode				
as_send	tid	signal				
as_return						
SEMAPHORES:						
sm_create	name	count	flags	&smid		
sm_ident	name	node	&smid			
sm_delete	smid					
sm_p	smid	flags	timeout			
sm_v	smid					
TIMERS:						
tm_set	timebuf					
tm_get	timebuf					
tm_wakeafter	ticks					
tm_wakewhen	timebuf					
tm_evafter	ticks	event	&tmid			
tm_evwhen	timebuf	event	&tmid			
tm_cancel	tmid					
tm_tick						

its *need to run* in a multitasking environment where many tasks may be ready to run at any moment. A task is given a priority at task creation time. A task's priority may be changed with the *t_setpri* directive.

A task's mode of execution is set up initially with the *t_start* directive, and may be changed using the *t_mode* directive. The mode of a task specifies its ability to be preempted, timesliced, to execute in *user mode*, to execute in *supervisor mode* with an optional interrupt level, and to disable/enable its asynchronous signal routine.

The *task manager* provides the pair of directives, *t_suspend* and *t_resume*, to control execution of another task.

A task is provided with a set of eight user defined and eight system defined software registers which may be set with the *t_setreg* directive, and read with the *t_getreg* directive. The software registers may be used for any purpose and may be especially useful for storing task specific information for use by shared libraries. Use of the software registers is not defined in the *RTEID*, but *system* software registers are reserved for use by shared libraries or other non-executive kernel functions. ***

Inter-task Communications

The *RTEID* executive supports communication and synchronization between tasks using messages and

TABLE 2. RTEID Directives (continued)

Name	Input Parameters					
REGIONS:						
rn_create	name	length	pagesise	paddr	flags	&rnid
rn_ident	name	&rnid				
rn_delete	rnid					
rn_getseg	rnid	sise	flags	timeout	&segaddr	
rn_retseg	rnid	segaddr				
PARTITIONS:						
pt_create	name	bsise	bnum	paddr	flags	&ptid
pt_ident	name	node	&ptid			
pt_delete	ptid					
pt_getbuf	ptid	&bufaddr				
pt_retbuf	ptid	bufaddr				
MEMORY MGMT:						
mm_l2p	tid	laddr	&paddr	&length		
mm_pmap	tid	laddr	paddr	length	flags	
mm_lmap	tid	laddr	flags	&aladdr		
mm_unmap	tid	laddr				
mm_pread	paddr	laddr	length			
mm_pwrite	paddr	laddr	length			
MEMORY MAP:						
m_ext2int	external	&internal				
m_int2ext	internal	&external				
MISC:						
v_return						
k_fatal	errcode					
DEBUG:						
db_control	tid	mode	qid			
db_remote	cpuid	requret	&rval	arg1	...	argN
db_block	tid					
db_unblock	tid					
db_getmem	tid	laddr	bufaddr	length		
db_setmem	tid	laddr	bufaddr	length		
db_getreg	tid	regnum	&regval			
db_setreg	tid	regnum	regval			
db_system	cpu	mode				
db_level	level	&plevel				
db_get_id	item_id	&ret_id	class	arg		
db_get_item	item_id	class	buffer	&sise		

events. Asynchronous communication is supported using signals.

Message queues support inter-task communication and synchronization. One or more tasks may send messages to a message queue, and one or more tasks may request messages from it.

Message queues are created at run time using the *q_create* directive. The creating task assigns a four byte name and *attributes* to a queue. The attributes define whether tasks waiting on messages from the queue will wait first-in, first-out (FIFO), or by task priority. The attributes also define whether the queue will limit the number of messages queued to a specified maximum, or allow an unlimited number of messages.

A message queue is identified by a name, assigned by the creating task, and by a message queue id (*qid*), assigned by the *RTEID* at *q_create* time. The *qid* is returned to the caller by the *q_create* directive, and must be used by tasks to send and receive messages from the message queue. Tasks other than the task which created the message queue can obtain the *qid* with the *q_ident* directive.

Messages are sent to the message queue from any task which knows the *qid*, using the *q_send*, *q_urgent*, and *q_broadcast* directives.

When a message arrives at a queue, it will be copied to one of two places. If any tasks are waiting at the queue, then the message is copied into the message buffer belonging to the first task. That task is removed from the wait list and is made ready.

If there are no tasks waiting at the queue, the message is copied into a system message buffer (the *RTEID* needs to maintain a pool of system message buffers for this purpose). This system message buffer is then entered into the message queue. If the message was sent using *q_send*, the message is entered at the tail of the queue. If the message was sent using *q_urgent*, the message is entered at the head of the queue. The *q_broadcast* directive sends a message to all tasks waiting at the queue, so they all become ready to run.

Messages are received from the message queue using the *q_receive* directive. When this directive is called, and a message is in the queue, the message is dequeued and copied to the task's message buffer. When no message is in the queue, there are several ways to proceed. If the calling task asked to wait, the task will be entered into the queue's wait list according the queue's attributes (FIFO or priority). If the calling task asked to wait with timeout, the task will be entered into a timeout list.

Message queues can be deleted using the *q_delete* directive. If any messages are queued, the executive will return the system message buffers to the system message buffer pool. If any tasks are waiting on the queue, then the *RTEID* will remove them from the wait list and make them ready. Waiting tasks will return from the *q_receive* directive with the *message queue deleted* error code.

A message has a fixed length of sixteen bytes. The content of the message is user defined. It may be used to carry data, pointers to data, or nothing at all.

Event Manager

Although inter-task synchronization can be accomplished using message queues, the *RTEID* also provides a second, higher performance method of inter-task synchronisation, called *events*.

Events are different from messages in that they are directed to other tasks, carry no information, and cannot be queued. A task may simultaneously wait for several events. In contrast, a single task may only wait on one message queue at a time.

Every task has the ability to send and receive events. Events are simply bits encoded into a event mask. Thirty-two events are available; sixteen will be reserved as *system* events and sixteen will be available as *user* events. A task can send one or more events to another task using the *ev_send* directive. The *tid* of the destination task is required as input, along with an event mask.

A task can receive events using the *ev_receive* directive. The events to receive are input to the directive, along with an option to wait on all of the events, or just one of them. If the events are already pending, then the event mask is cleared before returning. If the event condition cannot be satisfied, and the calling task asked to wait, the task will be suspended. If the calling task asked to wait with timeout, the task will be entered into a timeout list. Tasks that do not want to wait for an event must specify this option.

Signal Manager

Asynchronous inter-task communication is supported through the use of signals. Signals, like events, are simply bits encoded into a signal mask. Thirty-two signals are available; sixteen will be reserved as *system* signals and sixteen will be available as *user* signals.

A task can send one or more signals to another task using the *as_send* directive. If the receiving task has set up an asynchronous signal routine (*asr*) using the *as_catch* directive, the task will be dispatched to the signal routine.

A task may asynchronously receive signals by establishing an asynchronous signal routine to catch them using the *as_catch* directive. When a signal is caught, the task will be dispatched to the asynchronous signal routine. The signal condition will be passed to the task to enable it to determine what signals occurred.

The *as_return* directive must be executed to return the task to its previous dispatch address.

The sending task may optionally unblock the receiving task and unconditionally force it to execute the signal routine. Since this action is indicative of a fatal condition, the receiving task will not be allowed to return to the previous dispatch address with the *as_return* directive.

Semaphore Manager

The semaphore manager provides a set of directives to use in arbitrating access to a shared resource (many-to-one). The semaphore primitives provided can be used to fulfill different sets of requirements:

1. To control access to a single resource that is either available or not available, a task can

create a semaphore with an initial value of one.

2. To control access to a pool of n resources where at any moment m of those resources are available (where $0 \leq m \leq n$ and $n-m$ are not available), a task can create a semaphore with an initial value of n.

Arbitrating access to shared resources requires signaling when a predefined event occurs. Sophisticated synchronization may also require a *counter* to record the number of events sent, but not yet received, and a list of tasks awaiting receipt of the event.

The synchronization rules for semaphores are:

- The semaphore count is decremented by one when a task executes a *sm_p* operation. The task continues execution if the count is then greater than or equal to zero. If the count is less than zero, the task is put on a waiting list for the semaphore.
- The semaphore count is incremented by one when a task executes a *sm_v* operation. If the count is less than or equal to zero, the first task in the semaphore waiting list is placed in the ready state.

Time Manager

The *RTEID* time manager supports two concepts of time: calendar time and elapsed time. These functions depend on periodic timer interrupts, and will not work without timer hardware. The *tm_set* directive allows a task to inform the time manager of the current date and time (e.g., March 21, 1985; 12:04). The *tm_get* directive allows a task to request the current date and time from the time manager

(e.g., March 27, 1986; 09:24).

The *tm_evafter* directive delays a task for a specified number of clock ticks. When the elapsed time expires, the task is made ready.

The *tm_evwhen* directive allows a task to delay itself until a specific date and time. When the date and time arrives, the task is made ready.

The *tm_wakeafter* directive primes a timer event to occur after a specified number of system clock ticks. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

The *tm_wakewhen* directive primes a timer event to occur at the specified date and time. The requesting task is not blocked by this call. To receive the event, the *ev_receive* directive must be used.

The *tm_cancel* directive allows a task to cancel a timer event scheduled by the *tm_wakeafter* or *tm_wakewhen* directives.

The *tm_tick* directive allows a task or an interrupt service routine to inform the system of the occurrence of a system clock tick. This information is used to maintain correct calendar time, execute timeslicing, and decrement ticks from tasks which are currently being delayed or timed out.

Tick and timeslice are configuration parameters. A tick is defined to be some integral number of milliseconds. A timeslice is defined to be some integral number of ticks.

Memory Managers

The *RTEID* will support two different memory managers. A *region manager* provides allocation of variable sized memory *segments* or *regions*. A *partitic . manager* provides allocation of fixed sized buffers or *partitions*.

A *region* is an area of physically contiguous memory from which the executive can dynamically allocate *segments* to an application. A *segment* is a variable length block of memory.

A region is created with the *rn_create* directive. Like all objects managed by the executive, a region has a four character name, and, once created, a 32 bit region identifier *rnid*. Tasks other than the creating task can use the *rn_ident* directive to obtain a region's *rnid*. The directives *rn_getseg* and *rn_retseg* allocate

and return *segments* from a region.

When requesting a segment, if the request cannot be immediatedly satisfied, the requesting task optionally waits (with or without timeout) for a segment to become available. If it elects to wait, the task is placed in a memory wait queue associated with the region.

Each region has an associated *page size*, specified when the region is created. The page size must be a power of two. *Segment* lengths are always multiples of the page size. For example, if a task requests a 700 byte *segment* from a region having a 512 byte page size, a 1024 byte *segment* is allocated.

A partition is a pool of equal sized buffers. The *pt_create* directive creates a partition in a physically contiguous memory area provided by the caller. Like all objects managed by the executive, partitions have a four character name, and, once created, a 32 bit partition identifier *ptid*. Tasks other than the creating task can use the *pt_ident* directive to obtain a partition's *ptid*. *Pt_getbuf* and *pt_retbuf* allocate and return buffers from the partition.

Each partition contains a specified number of fixed size buffers. The number and size of the buffers is specified when the partition is created.

In a shared memory multiprocessor configuration, partitions may be shared between distinct processors. To do so, the creating task must declare the partition *GLOBAL* when it is created. If a partition is *GLOBAL*, then the *RTEID* will arbitrate access to the partition.

In a multiprocessor system, regions may not be shared between distinct processors. *Segments* may only be allocated or returned by tasks running on the processor from which the region was created. Hence, the *GLOBAL* flag used with the other create services is not supported by *rn_create*.

Memory Management Support

Hardware memory management support in real time applications has been largely ignored in the past; primarily due to performance limitations of most MMU's. With Motorola's M68030, this limitation is no longer true. This new processor has an embedded MMU that does not significantly impact the performance of the CPU. The M68030, for instance, contains an Address Translation Cache (ATC) which provides fast address translation of recently used logical addresses. On the M68030, the ATC is designed so that address translations are overlapped by other operations, thus reducing or eliminating performance penalties. The address translation occurs in parallel with on-chip instruction and data accesses before an external bus cycle begins. Without severe performance penalties, more and more real time applications will begin to take advantage of the protection and relocation features of MMU's. These features will improve real time software quality, reliability, and maintainability. Any real time executive that fails to support MMU's will be significantly handicapped in the applications of the 1990's.

The need for MMU's in real time applications will be significantly increased as programmers migrate to high level languages. For example, running sensitive real time Ada programs in a non-MMU environment is unwise and undesirable.

The *RTEID* optionally supports a memory management unit (MMU) such as the Motorola Paged Memory Management Unit (PMMU) found on the M68851 and M68030. An MMU provides memory protection, dynamic task loading and relocation, and dynamic memory allocation.

To provide these services, the executive adopts an MMU model which defines the *page size*, the structure and depth of the memory map, and the degree of control each task has over its own memory map. Different implementations of the *RTEID* are free to choose different models. However, the model chosen should allow the standard memory management services (regions and partitions) to operate in a consistent and intuitive manner in both an MMU and non-MMU environment.

Logically, the *RTEID* views a task's logical address space as a group of memory sections. Memory objects are mapped into a task's logical address space in variable size MMU sections. A single section is contiguous in the logical and possibly the underlying physical address spaces. Thus, the MMU is used to define a set of mappings for each task in the form:

(logical address, length) → physical address

Based on this model, the *RTEID* defines how the memory management services should operate, and defines additional services to manage the MMU directly.

MMU sections should not be confused with region segments. A *segment* is a block of memory allocated from a region. It can exist on any CPU. A *section* is only meaningful on a CPU with an MMU, and refers to a logically contiguous block of memory which is mapped into a task's address space.

When a task calls *rn_getseg* to obtain a segment from a region, the segment is automatically mapped into the task's logical address space at an *RTEID* assigned address. Since *rn_getseg* performs the mapping, the corresponding region is not mapped into the address space of other tasks using the region. This means that allocated sections are accessible only by the allocating task, and to those tasks that are explicitly given access to the segment. Thus, a segment is fully protected from inadvertent access by other tasks.

When a task executes a *pt_create* or *pt_ident* directive, the entire partition is mapped into the task's address space. Thus, tasks which share a partition can share and access any buffers allocated from the partition. Protection is on the partition level and individual buffers are not protected.

Dual Ported Memory

Dual-ported memory is commonly found in multiprocessor systems. In these configurations, the address of a particular byte of general purpose memory may be dependent upon the location of the task referencing it. In this case, a task must adjust the address according to the memory access requirements.

The *RTEID* provides a method for converting internal addresses to external, and converting external addresses to internal. This functionality acco mmodates the use of dual-ported memory and allows tasks to exchange addresses between processors.

The *internal* address is the address of a memory resource relative to the task which needs to access it. The *external* address is the address of a memory resource relative to a *remote* task which needs to access it.

Multiprocessor Support

More and more real time applications designs are pushing the limits of any single processor's capacity; thus forcing designers into multiprocessor designs. The *RTEID* provides simple and flexible real time multiprocessing. The *RTEID* model easily lends itself to both tightly-coupled and loosely-coupled configurations depending on the physical (hardware) configuration. Tightly-coupled systems are generally characterized by the ability to communicate over a backplane. By contrast, loosely-coupled systems generally use some external communications media, such as RS232 or network connection. Both forms of multiprocessing, tightly coupled and loosely coupled, have unique advantages, disadvantages, and suitability for a specific application. The *RTEID* does not favor one form over the other, but leaves this decision to the developer of an *RTEID* compliant executive.

The multiprocessing model of the *RTEID* defines each unique entity, such as a task, queue, event, signal, semaphore or memory segment as an object. Each object that may exist on a single processor configuration; or in a multiprocessor system. A *GLOBAL* object is accessible to any task regardless of which processors the object and task reside on. For example, the *t_create* directive has a *GLOBAL* flag which indicates whether the task is to be a global object or not. By setting the *GLOBAL* flag, the task identifier will be sent to all processors in the system to be entered into a global object table. The system is defined as the collection of interconnected processors; including processors connected by network or other communications media.

Leaving the details of the interconnection to the implementation of the executive frees the application code from unnecessary complexity. The task may simply refer to either a global or local object by its *id*. If the application developer wishes to synchronize tasks using message queues, for example, it does not matter whether the tasks all reside on a single processor, over multiple processors in a single backplane, or over multiple *crates* connected by a local area network.

For inter-processor communication, a simple m ssage queue is created with the *GLOBAL* flag set. In a tightly-coupled system, the resulting *glob l* queue is maintained by the executive and may be located in the shared memory area. But, since the details of inter-processor communications are hidden from the application task, a loosely-coupled implementation could be used with no changes to the application task.

Debug Support

During development of any software application, especially a real time application, it is critical that good development and debug tools are available. While it is possible to provide customized tools for every environment, it is highly desirable to be able to obtain *off the shelf* standard development tools which are capable of supporting a wide variety of host systems and targets. With standard development tools, the developer's time can be productively spent working on the target application; not on developing debugging tools or doing without.

The need for standard development tools available on a number of different hosts and from a variety of different sources implies the need for a standard interface between development tools, such as debuggers, and the real time target application environment. Such an interface is specified by the debug extensions to *RTEID*. By providing a standard interface to *RTEID*, a debugger can be easily ported from one compliant *RTEID* implementation to another.

The *debug manager* needs to be integrated into the executive for several reasons. Built-in interfaces provided by the implementer of the *RTEID* executive can efficiently access the necessary data structures and return the appropriate information without excessive overhead. Also, an integrated debug interface provides the ability to not only debug tasks, but also the ability to debug entire systems (including interrupt service routines). With the debug manager integrated into the *RTEID*, a debugger does not need to have utility tasks embedded in the target application. In addition to being more efficient, it also permits a debugger to set breakpoints in interrupt service routines without the danger of the debug service task becoming blocked.

There are three different parts of the *RTEID* debug manager: features for debugging tasks, features for debugging entire systems, and features for monitoring a running system. The *RTEID* debug manager provides a standard interface for the implementation of a debugger for *RTEID* applications. This debugger could be a task running on a single processor implementation of *RTEID*. In this instance, it would not be possible to perform system level debugging. The debugger could also be implemented as a board level monitor with knowledge of the *RTEID* implementation running on the processor. In this case, the monitor would be able to *freeze* the execution of *RTEID* applications and could perform system level debugging. The debugger could also be implemented on a second processor in a multiprocessor configuration. In this instance, the debugger could perform full system level debugging as long as multiprocessor communications are maintained.

Task Level Debug Features

The debug manager provides a set of services, which when combined with the other *RTEID* services, give a debugger the capabilities to control execution and access registers or memory of a target task.

The ability to debug a target task is established by the *db_control* directive. Once this directive is executed, the debugger manages the target task in several ways. It can examine and modify memory associated with the target task by using the *db_getmem* and *db_setmem* directives. It can examine and modify registers by using the *db_getreg* and *db_setreg* directives. It can also control the execution of the target task by issuing the *db_block* directive to prevent execution, and the *db_unblock* directive to continue.

When a debugger establishes control over a target task, a message queue identifier is passed to the *RTEID* by the debugger. Whenever a target task causes an exception, an illegal instruction for example, the target task is blocked and a message is sent to the debugger identifying the task and the exception. This feature can be combined with the other features of the debug manager to provide tracing and breakpoint capabilities.

System Debugging Features

Debugging a system is much more complex than debugging a task or collection of tasks. In order to debug a system, it should be possible to debug interrupt service routines (ISR's). Debugging ISR's causes several problems. For example, the processor interrupt mask must not be lowered outside of an ISR. Thus the system must not be allowed to execute tasks at a lower interrupt mask level while debugging the ISR. Additionally, an exception in an ISR may come at any time, and may occur when any task is executing.

In order to overcome these problems, debug control over a system is established using the *db_system* directive. This directive provides debug control over an entire system including all tasks and interrupt service routines on the specified processor. A mode parameter to the *db_system* directive specifies what type of control is desired. This parameter specifies what action is to be taken when an exception occurs in the system. Two types of action are permitted, *all* or *level*. When *all* is selected, then all processing (except debug commands) is suspended on the target processor until the debugger permits further execution. When *level* is specified, then task dispatching is modified so that only tasks and ISRs with an interrupt mask of at least the current level may run.

The debug manager also provides a *db_level* command that invokes the same task dispatching mode so that only tasks and ISR's with interrupt masks of at least the specified level are eligible to run.

System Monitoring Features

Part of the job of developing a system is tuning it for the best possible performance. A critical requirement when tuning a system is the ability to examine the data structures of the *RTEID* executive. The debug manager provides two interfaces to obtain this information in an implementation independent way.

The first directive is the *db_get_id* directive. This directive will return a unique identifier for an object in the system. It can be used to process all items of a particular class, or all items in a particular queue or list.

The second directive is the *db_get_item* directive. This directive returns information that describes a particular item identifier.

REAL TIME/UNIX INTEGRATION

Finally, to provide powerful standard development and runtime environments for *RTEID* users, the development and runtime environments must be integrated with a flexible and powerful operating system. Motorola has chosen to integrate its *RTEID* products with its UNIX operating system derivative called *SYSTEM V/68.*

A UNIX system's major fault for real time use is that it was designed for a multiuser, timeshared environment. Thus, it lacks non-blocking I/O, user controlled process scheduling, high resolution timers, high performance file systems, etc. It also tends to be less fault tolerant and may incur severe damage when crashing.

The advantages of a UNIX system are that it is standardized (to a degree), extensible, portable, full featured, and accepted by most engineers and developers.

While not fully standardized, UNIX systems provide multi-level standard platforms upon which to build. An AT&T published, three volume, System V Interface Definition, or SVID, defines a base system and several extensions. The base system defines a minimal kernel which is augmented by a variety of extensions. Future extensions may include real time services.

The extensibility and portability of UNIX are well known. Programs can be easily prototyped by using powerful features built into UNIX; such as the *shell.* Portability was built into UNIX intentionally. Written in C, UNIX can often be moved, or ported, to a new machine by replacing portions of a very modular kernel with code specific to that machine.

Examples of UNIX acceptance are easy to find. In our own hardware development area, engineers working on firmware under UNIX for less than a year cite the UNIX source code control utilities as extremely valuable; and use makefiles, archives, and shell scripts to great advantage.

Development Software

An off-the-shelf UNIX like Motorola's *SYSTEM V/68* provides many of the basic tools needed by a real time development team. Multiple users communicate by electronic mail and prepare specifications and design notes that may easily be put into publishable form. Source code may be maintained using the Source Code Control System (SCCS) or other available project management tools. Building target systems is possible by using the resident compiler, assembler and linker, or by using custom cross development compilers and assemblers. The process of building an application for the target system is greatly eased by programmer productivity tools such as *make.* Networking may link several systems for larger teams.

Beyond the tools supplied by UNIX, additional development tools are available. Due to the portability of applications written for UNIX, many independent software vendors are providing tools and utilities than can be easily ported to any UNIX development host. These tools include cross compilers and assemblers, high level debuggers, and Computer Aided Software Engineering (CASE) tools.

Custom tools can be built for a specific system using the rich resident tool building facilities under UNIX. Of particular interest to *RTEID* development will be languages and high level symbolic debuggers tailored to the embedded environment.

Modifications to UNIX can make it more useful during real time software debug. File system hardening minimizes or eliminates difficulties arizing from sudden system crashes that can be expected early in the development process of even the most carefully designed systems. Specialised drivers that connect UNIX applications to real time applications can be easily added.

Hardware Considerations

One disadvantage of using UNIX for development or during runtime is the load it puts on the system; especially *bus* contention. To avoid excessive loading, a *lite* processor can be used. A *lite* processor is one on which UNIX may be executed but which does not cause much load on the rest of the system.

The Motorola MVME147 processor is an example of a *lite* processor. It combines a Motorola M68030 microprocessor, an M68882 floating point coprocessor, four megabytes of memory, four RS232 serial ports, a parallel printer port, an Ethernet LANCE chip, and a SCSI controller, all on one double high VMEbus board. Attached to the SCSI bus may be hard disk, floppy disk, and tape units. A full featured UNIX system may be run on the MVME147 *lite* processor without impacting VMEbus activity. Thus, the MVME147 may be used to run UNIX in a real time system without hindering real time processors from providing the highest possible performance. Figure 2 shows a typical system configt ration. The real time processors *own* the bus. One or more of these processors may host the real time application along with whatever bus devices the application may require.

Processors like the Motorola MVME133XT can provide a high speed M68020 microprocessor with M68881 floating point coprocessor, four megabytes of memory, three eight-bit timers, a *time of day* clock, complete VMEbus interrupt and interface circuitry, and sockets for EPROM's. One or more of these real time application engines may provide complete ROM based real time solutions, or may be loaded and started by a *lite* UNIX processor in the same chassis.

With this type of hardware configuration, UNIX can perform disk I/O over the SCSI bus without affecting real time operations requiring VMEbus resources. Even network traffic can sneak quietly in the back door without disturbing real time activities.

Runtime Software

UNIX may be integrated with the runtime *RTEID* environment in three ways. The first way is to keep a full featured UNIX in the system. This UNIX can provide a user interface to the system, do background processing of data collected by the real time processors, and perform other general computing services. In addition, it may be used for development purposes.

The second way to integrate UNIX into a *RTEID* environment is as a server. This UNIX need not be a full featured UNIX; instead, it may be no more than a SVID compliant base system kernel, a couple utilities, and a user written application program. For instance, a machine control system may use a UNIX front end to store control patterns, accumulate production data, and interface with an operator.

The third way UNIX may be integrated with the *RTEID* runtime environment is by making UNIX based system services available to tasks running on the real time target processors. This can be done either by providing remote UNIX services, or by building a UNIX services layer on top of the *RTEID* layer, or both.

With a tightly coupled UNIX processor available in the chassis, remote UNIX services can be provided quite easily. For example, a real time task on a real time processor may be able to start a periodic operation on the UNIX processor. This activity may format and transmit an operation summary report to another node or another site.

The SVID base system definition specifies an extensive library of C subroutines, many of which are useful in a real time environment. Motorola has defined a real time library, called the *SVIDlib*, based on the SVID. It provides for both program and programmer portability. That is, programs that run on UNIX may be ported to the real time target with relative ease, requiring some modifications but not a full rewrite or rethink. Also, real time programs may be unit tested under the UNIX system without assembling and running target systems. More importantly for some, programmers familiar with the UNIX programming environment can use standard UNIX practices to solve many typical programming problems.

Even tasks that don't use UNIX features will probably require some I/O. Basic *RTEID* I/O services are provided by an I/O layer which provides *open*, *close*, *read*, and *write* operations in both synchronous and asynchronous forms. Initialization and device control operations are also provided. The

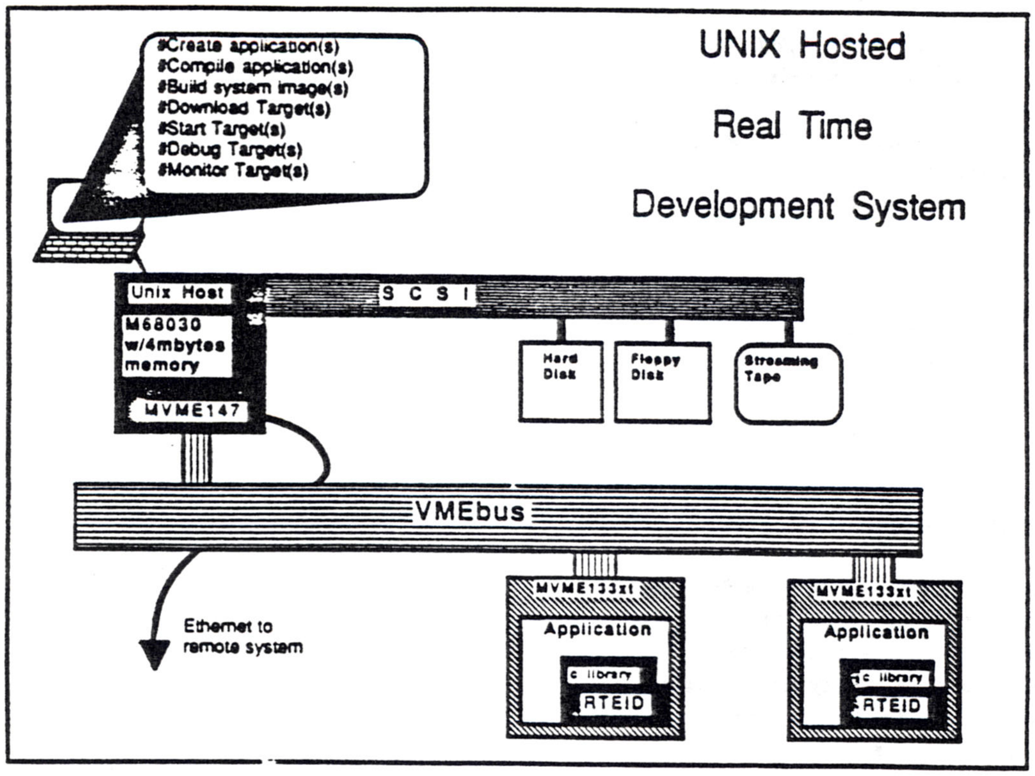

Figure 2. UNIX Based Real Time System

I/O layer also defines a standard interface for disk and terminal I/O. This I/O interface is independent of driver implementations.

A companion layer can provide network services as described by the SVID Network Services Extension. Still other imported extensions can provide a device independent terminal interface and sophisticated screen and ASCII window operations.

The model for integration of UNIX and the *RTEID* environment began by finding ways to get UNIX services available to tasks on real time processors. It is complemented by getting *RTEID* services into the UNIX environment. The integration may be accomplished in several ways. The simplest approach is to create a UNIX driver that provides a complete interface to the *RTEID*. To the real time processors, the UNIX system appears to be just another *RTEID* that uses *GLOBAL* objects. Since UNIX does not provide multiprocessor functions, but *RTEID* does, extending these features completes the integration.

CONCLUSION

Integration of UNIX and real time is the essence of Motorola's VMEexec program. VMEexec products will provide the services described above; including an *RTEID* compliant executive, the *SVIDlib*, and integration with *SYSTEM V/68.*

A real time services extension to UNIX is being developed by standards groups and AT&T. Eventually, a continuum of real time capabilities will be covered by a family of products that start with a uniprocessor real time UNIX and expand toward a distributed multiprocessor environment similar to the one described here. In all cases, the design of real time software needs to be standardized so that software investments are preserved. The *RTEID* described in this paper is one such interface. The SVID interface is another. The *SVIDlib* software library provides SVID compatibility in a real time environment; applications running on *SVIDlib* will be easily ported to a real time UNIX of the future.

TRADEMARK ACKNOWLEDGEMENTS

M68000, M68030, M68851, SYSTEM V/68, SVIDlib, and *VMEexec* are trademarks of Motorola, Inc.
UNIX is a trademark of AT&T.
POSIX is a trademark of the Institute of Electrical and Electronic Engineers, Inc.
pSOS is a trademark of Software Components Group, Inc.
MTOS is a trademark of Industrial Programming, Inc.
VRTX is a trademark of Hunter + Ready, Inc.
PDOS is a trademark of Eyring Research Institute, Inc.
C-EXEC is a trademark of JMI Software Consultants.

BIOGRAPHIES

Richard Vanderlin is Manager of Real Time Software for the System Software Department of Motorola Microcomputer Division. He is responsible for development of real time operating system software for Motorola's VMEbus board and system products. His previous experience includes design and development of software for the VERSAdos operating system as well as extensive work with real time control systems for semiconductor wafer processing.

Paul Raynoha is System Architect for the System Software Department of Motorola Microcomputer Division. His perspective is shaped by 20 years of hardware and software development experience from dedicated real time controllers to general purpose microcomputer systems. Mr. Raynoha is completing an MS in Computer Science at Arizona State University.

Dr. Brian Hansche is Manager of Development Software for the System Software Department of Motorola Microcomputer Division. Before joining Motorola, he was an assistant professor of Computer Science at Arizona State University. Dr. Hansche received a PhD in Computer Science from the University of Illinois. He is a member of ACM and IEEE.

Luke C. Dion is Manager of the System Software Department of Motorola Microcomputer Division. Prior to joining Motorola, Mr. Dion was Founder and President of PALOMINO Computer Systems, Inc., a UNIX porting and consulting company.