GA-A--20252

DE91 002553

**OVERVIEW OF
REAL-TIME COMPUTER SYSTEMS
TECHNICAL ANALYSIS OF
THE MODCOMP IMPLEMENTATION OF
A PROPRIETARY SYSTEM "MAX IV"
AND REAL-TIME UNIX SYSTEM "REAL/IX"**

by
J. CUMMINGS

Prepared under
Contract No. DE-AC03-89ER51114
for the U.S. Department of Energy

**GENERAL ATOMICS PROJECT 3466
DATE PUBLISHED: OCTOBER 1990**

GENERAL ATOMICS

MASTER

EB

AN OVERVIEW OF REAL-TIME COMPUTER SYSTEMS
TECHNICAL ANALYSIS OF THE MODCOMP IMPLEMENTATION OF A PROPRIETARY SYSTEM
'MAX IV'
AND REAL-TIME UNIX SYSTEM 'REAL/IX'

Author: John W. Cummings

Abstract

There are many applications throughout industry and government requiring real-time computing. Any application that monitors and/or controls a process would fit into this category. Some examples are: Nuclear power plants, Steel mills, Space program, etc. General Atomics uses eight real-time computer systems for control and high speed data acquisition required to run the nuclear fusion experiments.

Real-Time computing can be defined as the ability to respond to asynchronous external events in a predictable (preferably fast) time frame. Real-Time computer systems are similar to other computers in many ways and may be used for general computing requirements such as Time-Sharing. However special hardware, operating systems and software had to be developed to meet the requirement for real-time computing. Traditionally real-time computing has been a realm of proprietary operating systems with real-time applications written in FORTRAN and assembly language. In the past, these systems adequately served the needs of the real-time world. Many of these systems that were developed 15 years ago are still being used today. However the real-time world is now changing, demanding new systems to be developed.

This paper gives a description of general real-time computer systems and how they differ from other systems. However, the main purpose of this paper is to give a detailed technical description of the hardware and operating systems of an existing proprietary system and a real-time UNIX system. The two real-time computer systems described in detail are:

- Modcomp Classic III/95 with the MAX IV operating system.
- Modcomp TRI-D 9750 with the REAL/IX.2 operating system

Introduction

This paper contains several sections that describe the following:

- What is a Real-Time system?

This section describes what a generic Real-Time system is and how it differs from other Operating Systems. It describes why Real-Time systems are necessary and some of their applications.

- General description of an existing Real-Time system.

This section gives a general description of an existing Real-Time system that is widely used for many applications. It describes the hardware and software components that make up the system.

- Technical analysis of an existing Real-Time operating system.

This section gives a detailed description of the operating system used by the Real-Time system described in the previous section.

- New Real-Time systems.

This section describes the evolution of Real-Time systems and what is now available. In particular it will describe a new Real-Time Unix system.

What is a Real-Time system?

The text book defines a Real-Time system as a system that has well-defined fixed-time constraints. Processing must be done within the defined constraints or the system will fail. What does this really mean? Real-time computing can be defined as the ability to respond to asynchronous external events in a predictable (preferably fast) time frame. This requires balanced performance on three dimensions: instruction execution, interrupt latency, and I/O throughput. In addition to sufficient computational power (which is also required for Time-Sharing), Real-Time systems must provide efficient interrupt handling and high I/O throughput to meet the needs of the real-time application. Real-time computing, is not necessarily related to the time of day, though it could be, but rather is related to the time that an event occurs.

A Real-Time operating system must be able to respond to some internal or external event in a deterministic time frame, regardless of the impact on other executing processes. This is distinguished from a batch system, which may not have any time constraints, or a Time-Sharing operating system, which must share system resources (CPU, Memory, Peripheral Devices, etc.) equitably among a number of executing processes. In a properly configured time-sharing environment, no process should wait indefinitely for a system resource. In a real-time environment, critical processes must receive the system resources they need when they need them, without concern for the effect this may have on other executing processes.

The Real-Time system has to be considered as a complete system consisting of four components, the real-time operating system, system hardware, application programs, and the configuration of the components. The real-time operating system has already been defined. The system hardware requires the same hardware components as other systems (CPU, Memory, I/O, Peripheral Devices) as well as special hardware required for real-time computing. Special hardware is required that provides for external interrupts, hardware timers, analog and digital interfacing, and interfacing to specialized devices (Instrumentation). Unlike other systems, the application programs must be included as part of the Real-Time system. This is because the operating system allows a process to take whatever resources it needs which can lock out other processes. The configuration of the components in a Real-Time system is a very critical design parameter. It is critical that the operating system, system hardware, and application programs be designed and configured so that they all play together. This requires each system to be configured for the application it is used for.

Traditionally, real-time computing has been a realm of proprietary operating systems, with real-time applications written in assembly language or FORTRAN. The operating systems were designed for performance and not for 'User Friendliness'. They provide only minimal protection for processes and do not provide a lot of the elegant services found in a typical robust time-sharing operating system. Portability of application programs is usually limited to systems from the same vendor and maybe not even that. In the past, the real-time environment has not demanded elegant operating systems, so the vendors were not pushed to enhance their operating systems other than for performance.

Real-Time systems are typically used for data acquisition and process control applications. There are basically two types of processing, continuous and batch. Some of these applications are: controlling an industrial plant (i.e. Steel Mill), monitoring and/or controlling a nuclear power plant, space program, etc. We use them at General Atomics for the high speed data acquisition and control in running our nuclear fusion experiments. The need for a Real-Time system can be demonstrated in the monitoring and controlling of a Nuclear power plant. Typically the system monitors the plant operation by doing high speed data acquisition and controlling the plant's operation based on the data acquired. The data is acquired by reading from the I/O interfaces to instruments and various input signals that monitor the plant. The data is processed by the computer resulting in selected control signals being output through the I/O interfaces to control the plant's operation. The application programs that acquire the data and output the control signals must be able to respond to the data in a predictable manner. For example, if an alarm were to occur, the system must know about it as soon as possible and output the appropriate control signal(s).

Description of an existing Real-Time System

The Modular Computer Systems (MODCOMP) Classic computer series will be used as the example of an existing Real-Time system. There are two reasons for this choice. The first reason is that I am very familiar with the system. I have thirteen years of experience with the Modcomp computer systems. The second reason is that Modcomp is one of the leading vendors of real-time systems. Some of the other companies providing Real-Time systems are; Gould (Encore), Harris, Masscomp, Hewlett Packard, DEC, etc.

Modcomp supplies both 16-bit and 32-bit systems which run the MAX IV/32 real-time operating system. The systems are functionally similar. The Modcomp Classic series is a family of computers with 16/32-bit CPU's that run the MAX IV operating system. The 16-bit Modcomp Classic III/95 will be described as it provides the highest performance levels of any computer of its class. Modcomp now has a 32-bit machine that runs a Real-Time UNIX operating system. It will be described in a later section of this paper.

It is necessary to describe each of the four components, hardware, operating system, application programs, and configuration that make up the Real-Time system. The ability of the system to perform real-time computing is dependent on all of the components. For example the CPU contains hardware, not normally found in other systems, that provides for fast context switching.

The following is a general description of the components of the Modcomp Classic real-time computer system:

Hardware:

A fully configured Classic III/95 is comprised of a CPU, 4 Mega-bytes of SRAM, and two dual I/O processor (IOP) boards. It may also be configured with an optional high speed Communications Processor (CP) that significantly increases performance in communications applications. Performance specifications for the Classic III/95 are:

• Execution speed	- 2.5 MIPS (Whetstone 1 benchmark)
• CPU cycle time	- 67 ns.
• Interrupt handling	- 320 k/sec.
• I/O bandwidth	- 8.8 Mbytes/sec. aggregate.
• Context switch time	- < 4.0 usec.

CPU

- Two modes of operation, privileged and non-privileged providing control of a programs memory access rights and instruction execution. A user program may operate in non privileged (default) or privileged mode (dangerous) mode.
- 362 conventional machine language (assembly) instructions, 1, 2, 3, or 4 16-bit words long, classified as privileged or non privileged. Memory operand addressing may be by bit, byte, word or multiple words (1-8) with 16-bit virtual addressing. There are special extended addressing instructions that allow for addressing of the full 21-bit physical addressing space (privileged mode). I/O and map selection instructions are privileged.
- High performance pipeline instruction unit that allows for concurrent instruction execution and operand fetches.
- On board high speed floating-point processor. The floating point circuitry improves computational power by performing operations in 32-, 48-, and 64-bit operands.
- 128 sets of 16 general purpose registers. This large number of general purpose registers enables context switching among several tasks without time consuming register saving and restoring.

MEMORY

- High speed solid state static memory (SRAM) configurable in 2 or 4 Mbytes capacity with 2 or 4 way interleaving.
- Accessing of instructions or data in one machine clock cycle (67 ns CPU cycle). This allows the CPU to run at the maximum speed for the entire application.
- Five concurrent memory access paths that provide simultaneous CPU and I/O access to the main memory, that optimizes utilization of available memory bandwidth.

Memory Management

- Memory is organized into hardware memory maps to provide hardware "relocation" capability of nonresident programs and to avoid memory fragmentation. These maps can be thought of as though they are separate 64K words memory areas. One map can be divided into four 16K words memory areas for executing smaller programs. In this mode, the small maps are fully isolated from each other, and address references above 16K words are trapped.
Actually, these maps do not represent physical memory areas at all, but are switchable addressing ranges. These addressing ranges are sometimes called virtual addressing spaces. The large maps are addressable from 0 - 64K words, and smaller maps are addressable from 0 - 16K words.
Actual memory is configurable to 8192 pages of 256 (512 bytes) words per page. Actual memory pages are not all necessarily accounted for in the maps. Some memory pages may even be duplicated in each map (for example, global shared memory areas). These maps are "memory address patch-boards" that allow any physical memory page, located anywhere in actual memory, to be addressed by any convenient 256-word range of a program's virtual address range.
- Eight by 256 hardware register map files that enables several programs to remain mapped concurrently. This provides very fast task switching, as well as reducing system overhead to a minimum.

- Virtual memory management hardware that converts a CPU instruction's 16-bit virtual address into a 21-bit physical address. The 21-bit physical address provides access to the full 4 Mbyte physical address space.

PROGRAM REGISTER OR ANY VIRTUAL EFFECTIVE ADDRESS

INPUT / OUTPUT

- I/O processor (IOP) that provides two high performance I/O busses. Each I/O bus provides 2.2 Mbytes of throughput. Can be configured with two IOP's providing 4 I/O busses for an aggregate throughput of 8.8 Mbytes. Each I/O bus has its own direct memory access path to main memory.
- Optional Communications Processor (CP) that is a variation of the IOP. The CP provides one standard I/O bus plus a special communications bus. This provides a direct memory access multiplexed data path for up to 512 channels between main memory and Modcomp's Universal Communications Subsystem. This significantly increases performance in communications applications.
- 16 prioritized hardware interrupt levels that provide for 5 user external interrupts, multiprocessor communications as well as the standard I/O interrupts and system traps.

- Support for all standard I/O devices (i.e. Disks, Printers, Terminals, etc.)
- Special hardware controllers and subsystems that connect to the I/O bus. Modcomp supplies controllers and subsystems for interfacing to analog and digital inputs and outputs, custom user interfaces (RTU's, etc.). These all connect to the standard Modcomp I/O bus.

Operating System

The operating system is the MAX IV real-time operating system. There is a networking subsystem available, MAXNET Distributed System, available as an extension of MAX IV. MAXNET links multiple computers together in a distributed network configuration. MAXNET will not be included with our system.

MAX IV is a real-time, communications-oriented, multitasking, multiprogramming operating system design to run on the Modcomp 16/32 bit Classic series computers. It is designed for real-time processing and a variety of communications applications. MAX IV can manage the execution of a combination of real-time and batch processing programs. It provides a set of utility services and high-level language support processors each with extensions to the standard ANSI, IEEE, ISA, BSI, for the implementation of real-time applications. MAX IV may be configured in a variety of forms to support a wide range of general purpose and dedicated user applications.

General Description

- Event driven preemptive priority scheduling. Optional round robin scheduling for tasks at the same priority level.
- Dynamic allocation of system resources.
- Optional pre-allocation of system resources.
- Fast task loading.
- File and data management processors.
- Automatic roll-in / roll-out.
- Alternate device switching.
- Intertask communication (ITC).
- Operator communications.
- Interactive batch program development facilities.
- Intelligent Processor System (IPS) support.
- Extensive Utility library with special utilities for real-time applications.

Application Programs

The application programs must be considered as a component of the real-time system as they can affect the operation of other processes, process scheduling and the allocation of system resources by the operating system. A properly designed time-sharing system does not allow this to happen. A time-sharing system tries to give equal time and resources to all of the processes running on it. One users application program executing does not affect another users program. This is definitely not the case for a real-time system.

MAX IV allows for running the system in a time-sharing mode like any other multi-user system, however any process may take over the system or affect the execution of another process. This is obviously going to terminate time-sharing while the process(es) are executing.

The applications programmers must consider many things when they design their program. They must consider, system resources (dedicated or not) required, task priority level, and the general effect that the process will have on other processes.

Configuration

The configuration of a real-time system is another very important consideration when designing a real-time system. The configuration of a time-sharing system is usually dependent only on the number of users supported. A real-time system requires configuring the hardware, operating system and application programs in a manner consistent with the application.

Some real-time applications are single dedicated processes that require that the system monitor and/or control a single process. Others require the real-time system to monitor and control several processes at one time such as controlling a power the operation of a power plant. Modcomp systems are typically employed in applications requiring total process control of all operations (i.e. nuclear power plant).

Some of the considerations that are unique to a real-time system are:

- Connection of the hardware external interrupts between the events being monitored and the user tasks.
- Selection of the task scheduling algorithm (done at SYSGEN time).
- Allocation of software timers.
- Allocation of MAP 0 space. MAP 0 contains the MAX IV nucleus and other tasks that are put there. It is easy to run out of MAP 0, so careful consideration must be given to what gets placed in MAP 0.

Technical Description of MAX IV

The following is a description of the MAX IV operating system functions:

System Generation

The system generation process (SYSGEN) requires the user to write several specification statements in a higher-level MACRO language. These statements permit the user to customize the MAX IV system including only those optional elements that are required for applications. Performing a SYSGEN is done in four phases.

Phase 1 -- Specify the desired resident elements in four parts with basic MACRO languages:

- a) System Structure Elements
- b) Basic I/O System Elements
- c) Task Related Elements
- d) File Manager Configuration

To these special assemblies, other user-specified elements can be supplied (Resident EXecutive (REX) Services, operator directives, resident task programs). The resulting group of object programs can be thought of collectively as the "main program" of the operating system and is called the SYGBLOCK. In most cases, these program elements consist of only simple data bases and linkages that "call" the appropriate elements from the MAX IV System Elements Library.

Phase 2 -- Link the object elements produced in Phase 1 with the elements of the MAX IV System Elements Library, loading only the routines that were referenced.

Phase 3 -- Load and initialize the system object image in main memory. A special Stand-Alone Loader (SAL) can be used to perform this function.

Phase 4 -- Use the newly generated system to establish all nonresident elements.

Interrupt routine, tasks, and overlays

MAX IV is an interrupt-driven operating system. Sixteen possible hardware interrupt levels activate memory resident executive routines that monitor the execution of system and user programs. These executive resident routines are referred to as directly-coupled interrupt routines.

Tasks are the basic units of multiprogramming in the MAX IV environment. A task is a complete program that includes an additional description of resources which permit it to execute in a manner that is concurrent with, but independent of other program elements. The description of resources are defined at the time that the task is cataloged on the disk as a load module. Tasks run at software generated priority levels below the lowest hardware interrupt level. There may be up to 256 unique priority levels, each with multiple task assignments.

An overlay is a program which runs using the resources of the task it overlays. An overlay may reassign the resources of the task or assign additional ones to be used by the overlay. A program executing as an overlay to a task runs at the priority level of the overlaid task.

In the MAX IV environment many tasks compete for central processing unit (CPU) time. The TASKMASTER determines the next task to be executed, and assigns all the necessary resources from the available pool, or re-allocates them from the lowest priority task awaiting execution. The taskmaster routine is a directly-connected interrupt routine triggered by the lowest hardware interrupt level. The following is an illustration of the hardware interrupt levels and the hardware/software priority structure.

Hardware Interrupt Level	Interrupt Signal	MAX IV hardware/software priority structure
0 1 2 3 4 5 6 7 8 9 A B C D E F	Power Fail Safe/Auto Start Memory Parity System Protect Multiprocessor Communications Unimplemented Instruction Trap Floating Point Overflow Real Time Clock External External External External I/O Data Party Line I/O Special Party Line Console Interrupt Task Scheduler	<p>INCREASING PRIORITY</p> <p>0 F 0 255</p> <p>HARDWARE PRIORITY LEVELS</p> <p>SOFTWARE PRIORITY LEVELS</p>

Task Scheduling

The MAX IV performs task scheduling by the standard system context switching routine (TASKMASTER). The TASKMASTER context switches are event driven. There is a SYSGEN statement for the TASKMASTER that influences how context switching is performed. The TASKMASTER statement defines a period of time as an event. If no natural context switch occurs during the specified time interval, one will be forced on expiration of a directly connected timer. The timer is reset by the TASKMASTER on event scans.

Normally, the TASKMASTER processes tasks strictly on the basis of priority. Even if several tasks exist in the CPU queue (CPUQ) in clusters at the same priority level, the tasks closer to the head of the CPUQ will actually be higher in priority, than tasks of the same priority level that are lower in the CPUQ. Specifying a ROUNDROBIN scheduling algorithm for tasks of like priority level, results in a more even distribution of computer time to tasks of equal priority when they each have a tendency to become "compute bound". This algorithm moves the task using the most CPU time lower in the CPU queue. Task movement in the CPU queue is with respect to priority clusters. High priority tasks in the CPU queue are always given preference over lower priority tasks regardless of CPU utilization. The following is the SYSGEN TASKMASTER statement:

TASKMASTER < rate > < ROU(NDROBIN) > < ROL(ONHOLD) > < Interval >

- rate** - Specifies a period of time that is allowed to pass before an event scan is forced by the real-time clock. If 0 is specified, no forced context switching occurs and only natural events drive the system's context switching function.
- ROU(NDROBIN)** - Optional parameter that specifies that the round-robin scheme of scheduling is used for groups of tasks in the CPUQ at the same priority level.
- ROL(ONHOLD)** - When specified along with parameter 2 (ROU), this parameter rolls out tasks that are suspended or are in I/O HOLD or OC HOLD. This parameter only affects batch tasks.
- Interval** - Specifies the the number of basic clock interrupts in the interval to be used by the ROUNDROBIN algorithm to rearrange the CPUQ.

MAX IV tasks can be scheduled to be activated, suspended, resumed or killed with the use of timers or external interrupts. Timers may be used as one-shot timers which automatically return to the pool once they expire and accomplish their scheduling function. They also may be periodic timers which must be deallocated. Timers can be scheduled to perform their function according to elapsed time (relative to when they are set) or according to time-of-day.

A single delay timer is also available to each task in the system. Unlike the timers in the system pool, individual task delay timers can only resume the task that possesses the timer and cannot be made periodic.

External interrupts may be used for scheduling according to an event, regardless of the time. Unused external interrupts may be specified at SYSGEN time for this use, and a re-entrant interrupt service routine can be connected to each designated level. Such levels may then be allocated to a non-resident task as a task scheduler by the connect service (connect timer). The scheduled task will be activated, suspended, resumed or killed at its software priority level when the corresponding interrupt level is requested (externally by a hardware signal or internally by a set interrupt request instruction). These schedulers can also be made one-shot or periodic.

Finally, one task can schedule another task directly by requesting that it be activated, suspended, resumed, or killed; or the task can connect another task to a timer or external interrupt. Timers may be connected, unconnected, thawed or frozen by any task at any time the task wishes to do so. Also a task can use the ITC service which is described in the next part.

Intertask Communications (ITC)

ITC is the MAX IV message service (MS). It is a system module that allows for variable length collections of data to be transmitted between tasks. Messages between tasks flow along distinct communications paths known as virtual circuits. When two tasks by prior agreement open a virtual circuit, both tasks may send and receive messages along this temporary bi-directional path. Messages are queued on each virtual circuit. A task may open multiple virtual circuits between itself and other tasks and may also have several distinct virtual circuits open between itself and a single other task. A virtual circuit may be cleared at anytime by either task, thereby ending communications. Optionally, a task waiting for information that has not been transmitted can suspend itself until the message arrives. Thus each task may use the MS for synchronization as well as information transfer. A task may activate another task by opening a virtual circuit to it and sending a message. Tasks using the message service must be declared as servers for ITC. This is done by creating special ITC data structures for each task wanting to use ITC. This is done at either SYSGEN time or may be done interactively through special ITC system routines. Operator communication directives can be used to dump information stored in the various ITC data structures, thereby generating information about messages queued on one or more virtual circuits.

Memory Management

The allocation of actual memory pages to pages of virtual addressing space is managed by certain Executive services. Any number of actual memory pages (512 bytes per page) scattered throughout actual memory may be put together to form any required contiguous virtual space for use by a program. Because this feature eliminates a common memory fragmentation problem, such techniques as multiple memory pools and fixed memory partitions are unnecessary compromises for MAX IV. A single Available Memory Pool (AMP) is all that is necessary for MAX IV to service the many asynchronous demands for memory space.

Associated with an executing task is a Program Status Doubleword (PSD), which is related to the program counter and program status registers in the CPU. The PSD contains two 3-bit fields that specify which map is to be used for instruction addressing and which map is to be used for operand references. The map used by the task for instruction fetching is called the INSTRUCTION MAP (IM), and the map used for operand addressing is called the OPERAND MAP (OM). All words of instructions are fetched from the addressing space defined by IM. All operands and indirect address constants are fetched from the addressing space defined by OM. Regardless of whether or not the task actually uses one or two maps when executing, the contents of these fields are always used to select a map for the translation process. Both fields contain the same map number for tasks using only one map.

Tasks located in MAP 0 are permanently resident and privileged, and are loaded with the MAX IV nucleus. All other tasks are dynamically allocated. MAX IV constructs contiguous virtual addressing space from randomly distributed pages of actual memory. The number of simultaneous tasks configured in the system is limited only by memory size.

Memory is dynamically allocated on a page basis (512 bytes per page) to each active task as it is loaded. These allocations are made on a single- or multiple-page basis and are recorded in the task's map image. Active tasks can request additional memory at run time. When individual tasks exit, memory is automatically de-allocated. Memory MAPS 1, 2, and 3 are dynamically assigned to large tasks, and MAPS 4, 5, 6, and 7 are dynamically assigned to small tasks, as tasks become active or interrupt each other. These tasks, or overlays, are normally positioned at the virtual memory address 0 of their assigned map by a linking process. Each such task treats a full 64K (or 16K) of virtual addressing space as if it were the only program in a computer of that size. Once a task's memory allocations have been loaded into a hardware map, it is probable that they remain until the task finishes execution. However, if six (or more) other tasks interrupt it during execution, its memory map may be reassigned while interrupted. When the interrupted task is resumed, the scheduler (TASKMASTER) retransmits its memory allocations to a hardware memory map made available to it at that time. A Map Image (MI) of each active task is contained in its Task Control Block (TCB) extension so that the system can retransmit the task's memory allocation information to a hardware map when necessary. This Map Image is also used directly by the I/O handlers and DMP's (Direct Memory Processors), to assist them in passing data into and out of the actual memory assigned to a task, even though the task may not currently have its image contained in the hardware map. The TASKMASTER has the ability to assign a task to any of the MAPS 1 to 7 regardless of program size if its normal size map is not available.

Memory maps never need to be saved when a task using one is interrupted, because a copy of each task's Map Image (MI) is maintained in memory as part of its Task Control Block (TCB). Memory map images need to be loaded into hardware maps only when an interrupted task resumes execution and its map was reassigned to a higher priority task while it was interrupted.

Tasks may be classified as Two Map Tasks if they will not fit in one large map. The hardware permits unique map assignments for both instructions and operands that allows a program to be assigned to two available memory maps. This allows for the tasks' instructions to be in a different map than its operands. The map assignments for the instruction and operand maps are to the same map for one map tasks.

One important point to remember is that a program can be classified in such a manner that it will take all of the system's resources (memory, CPU, I/O, etc.) that it needs and will keep them as long as it is executing. This ability for a task to dominate the system is a critical requirement of a real-time system.

The System Loader and Load Modules

Nonresident overlays and tasks are cataloged on disk in a special load module format that permits these programs to be brought into memory in large single blocks, often requiring only two disk accesses. The load module is in a format that requires no software relocation (if the program has been cataloged in a normal manner with the MAX IV Task/Overlay Cataloger (TOC). The loader utilizes the natural hardware relocation benefits of memory mapping. The absolute locations of all nonresident programs, global commons, and reentrant shared libraries are determined at link-edit time or cataloging time. For special applications, a module may still be cataloged so that it can be relocated at load time, but this is rarely necessary in a MAX IV system. Each load module of a root task or overlay consists of two types of records called Load Program Records (LPR's) and Module Object Records (MOR's). The "average" program has one LPR (a disk sector) and one MOR (a multi-sector record). Pointers to each of these items are included in the Load Module Directory (LMD) of the cataloged program load module file.

A Load Program Record (LPR) contains information (called "resources") for the program to be loaded. The system uses this information to select the appropriate memory and space environment of the program to be loaded (if not already allocated). For a root task, the LPR also contains the task's resource assignments that tell the Load Module Retriever how to build a Task Control Block (TCB) for the task when it is activated and becomes an independent multiprogramming entity. The LPR also contains a "string chain" that specifies to the loader how it should read the body of the program. This chain facilitates the "scatter" read of the program even when not contiguous in memory. The chain is in a format that is directly interpretable by the Input/Output Processor of the system hardware.

The body of a nonresident program consists of one or more program extents called Module Object Records (MOR's) that are defined in the LPR's. These MOR's are read into the program's allocated memory area using a scatter-read technique as specified by the string chain in the LPR's. Only one disk access is normally required to complete the transfer.

All load module files have their directory of cataloged programs as the first data item on the file. For critical program storage files, a resident directory may be defined in the system nucleus, which keeps resident copies of all or part of the file's load module directory for immediate access by the Load Module Retriever.

Automatic Checkpointing -- Roll-In / Roll-out

The ability of a task to retain its main memory resources is determined by the task's execution priority level. Since batch-processing tasks and real-time tasks under development execute at lower priority levels, their main memory resources may be taken away at any time, if higher priority tasks need these resources. As a general rule, batch-processing tasks should be executed at the lowest priority levels to ensure that they are the "first-to-go" when memory becomes crowded.

When any task needs memory and no memory can be found in the Available Memory Pool (AMP), then the "memory allocator" queues the request to the Roller task. This task attempts to find the lowest priority task that is designated as being rollable and has enough memory pages to satisfy the demand. The Roller executes as a resident system task (ROL) but has the ability to move its effective execution priority level in the CPU queue to a level appropriate to service the highest priority task needing memory. The roller task then rolls-out the lowest priority rollable task to disk, freeing up enough memory for the higher priority task needing memory. When enough memory becomes available, then the task that was rolled out to disk is rolled back into memory. When the Available Memory Pool is capable of serving all task memory requirements, the ROL task moves itself to the bottom of the CPU queue, and stays there until some task needs more memory than is available in the AMP.

The system also provides for manual checkpointing. The ROLL REX service can override the above mentioned "Automatic Checkpointing" algorithm. Using this service a named task can be requested to be rolled out of main memory or to be rolled back into main memory. It can also change the rollability of a task. When a task is requested to be rolled out of main memory, it stays rolled out until a REX ROLL (In) service request occurs.

There is a pool of memory pages called the Emergency Memory Pool (EMP) managed by the roller task. This pool is available to provide memory pages on an emergency basis to a task that needs only one page. If a task needs a single page and it is not available in the AMP, then ROL will assign one page from the EMP to the task if a page is available. This saves having to roll a task out if the task requesting memory needs only one page. ROL will automatically place available pages back in the EMP, maintaining the predetermined number of pages in the pool.

Hardware Context Switching

Hardware context switching is a term used to describe the program switching action of a hardware interrupt. A program is interrupted, another process (the interrupt subroutine) is executed, and the original program is then resumed. The context switching time is usually only a measure of the entry time and exit time of the interrupt subroutine, but must practically include the saving of all machine states (usually registers), that are necessary before the interrupt subroutine can do useful work, and the restoration of these states before the interrupt subroutine can return to the interrupted program. In the CLASSIC system, it is not necessary to save the registers because of the presence of the hardware context file. This significantly reduces the hardware context switching time. If an interrupt needs a whole block or part of a block of 15 general registers for its execution, a unique general register block (a section of the hardware context file) is dedicated to the level. If only a few registers are needed by an interrupt subroutine, a general register block can be assigned that is shared by with other interrupt levels, by mutual consent. Not only are time and memory saved by not having to save the registers needed by the subroutine, but additional time is saved by letting many of its most important pointers and data values remain in registers, from one interrupt execution to the next. The additional time necessary to reload these arguments from memory on each interrupt is avoided, making these general register blocks similar to a high-speed memory extension.

Interrupt context switching also involves saving and restoring two additional CPU registers, the PR (Program Address Register) and the PS (Program Status Register). Together they are referred to as the PSD (Program Status Doubleword). Four dedicated memory locations, organized into two pairs of words, are assigned to each unique hardware interrupt level for this purpose.

The NEW PS word of the interrupt has fields that select the states such as, general register block, and memory map(s) to be used by the interrupt subroutine. When the general register block field in the OLD PS word (register file being used by interrupted program), differs from the general register block field assigned to the interrupt subroutine (NEW PS), the automatic register file swapping occurs. This feature does not penalize the user by wasting time, since it can take place in essentially the same time that is required to access the four memory cells necessary to effect the interrupt entry. This swapping process is reversed during interrupt exit, but only two memory cycles are required. This provides for very fast hardware context switching.

MAX IV uses this capability to effectively reduce its own system overhead. Its most frequently used interrupt levels (TASKMASTER, Clock Interrupt, Data Interrupt Party Line, Service Interrupt Party Line) are assigned complete or partial dedicated register blocks. The most important pointers, addresses, and constants used by these subroutines are maintained in the dedicated registers and may not even exist in memory (except at system start). Other dedicated register blocks may be shared by the remaining interrupt levels.

Since the swapping of the PS words upon interrupt entry and exit also affects the assignment of memory maps (one for instructions and one for operands), an interrupt routine may automatically be made to use a different map than the program it interrupted. For most hardware levels, this new map is always selected to MAP 0. Thus, all interrupt subroutines and their dedicated memory locations are mapped into the addressing space of the system map (MAP 0). When an interrupted program is resumed, its memory map(s) is (are) automatically reselected.

Software (TASK) Context Switching

All user tasks and system tasks operate at "software-generated" interrupt levels created by the TASKMASTER routine that operates at the lowest hardware interrupt level (#F). Since this interrupt is the only level that always "knows" that it is interrupting the hardware background (non-interrupt) time of the CPU, it can effectively multiplex this hardware background time into many additional pseudo-levels, called task levels. The TASKMASTER has its own dedicated general register block, in which it maintains the most important pointers (current TCB pointer, masks, pointers for available register blocks or Memory Maps) used for software context switching. The program it interrupts is always the currently executing task. The program it resumes is either a return to the original interrupted task or to another task that it chooses to run instead of returning to the interrupted program. Since the TASKMASTER can manipulate the "old PS" word of the interrupted program, it can assign surplus register blocks (those register blocks not dedicated to hardware interrupt levels) and surplus memory maps, to any task when it transfers control to it (TASKMASTER interrupt exit).

If the number of tasks in the system is less than the number of available register blocks, software context switching can be accomplished without any general register files having to be saved and restored by the TASKMASTER.

In more complex applications, where the number of concurrently active tasks is greater than the number of surplus general register blocks, the TASKMASTER must save and restore registers when (or if) required. Even in these system applications, however, the average number of nesting levels is usually such that register saving and restoring is rare. Thus, another major contribution to operating system overhead can be reduced.

Usually 12 blocks of surplus general registers exist for assignment by the TASKMASTER. As tasks become active, each is assigned its own unique register block. As tasks become active (user tasks or system tasks), the needed register blocks are taken from the lower priority interrupted tasks (if no surplus blocks are available). The register block contents of such tasks are saved in memory before they are reassigned to the higher priority tasks. If any task exits (terminates execution as opposed to simply relinquishing time), its register block is returned to a pool, so that it can be assigned quickly to new tasks as they become active. If an interrupted task is reentered and its register block has not been reassigned while interrupted, then it is simply "switched on" by the normal restoration of its PSD instead of having to be restored from memory.

If only one general register block is used by the TASKMASTER, this single block has to be saved in memory (in the interrupted task's Task Control Block (TCB)) and restored from memory for each task interruption.

Basic Input / Output System

The MAX IV Basic I/O System (BIOS) consists of two parts:

- Interrupt-driven or DMP-driven handlers (drivers) exist for all physical I/O devices. These handlers are directly-connected to the I/O channels they control. The user need not interface to them directly.
- A centralized Basic I/O System, addressable as a series of Executive Services (REX calls), allows a user to interface to device handlers indirectly, by way of logical files. Logical files are assigned to devices at task cataloging time, or at run time. These services permit a programmer to talk to the file's assigned device only in terms of physical records (i.e. card, block, or sector).

I/O Handlers and Devices

Devices may be real, logical (subdivisions of a disk medium such as real files), or imaginary (simulated by a task called a symbiont). Some devices have their own independent controller channel, while others must share a common controller channel. All devices are identified by name in the Logical Device Table (LDT).

The interrupt-driven handlers permit I/O operations to be performed concurrently (overlapped) with the execution of the program that request the operations. Slow devices use data interrupts to transmit bytes/words, higher speed devices use Direct Memory Processors (DMP's) to reduce interrupt overhead.

Service interrupts are used by all devices for initiating and terminating all operations. These interrupts manage the queue of operations waiting to be started on a particular controller channel.

Device error recovery may be automatic (retry initiated by service interrupt with unrecoverable errors reported to the system operator), or may be under complete control of the calling task. The latter method allows a critical task to perform suitable application-dependent error recovery techniques, including device substitution.

Device handlers have both device-independent and device-dependent modes of operation. The former mode lets the handler simulate (if necessary), a series of device characteristics that are common to a class of similar (but non-identical) devices. The latter mode allows a programmer to use the unique characteristics of a particular device if desired.

Unusual devices, which have difficult or unusual control protocol, or perform functions unavailable in real devices, may be interfaced to the system by a symbiont. These symbionts are simply tasks that present an imaginary device interface to the programmer while insulating the user from the peculiar protocol of the real device or communications channel. The printer spooler/despooler is an example of a symbiont.

Basic I/O System Services

The calling task has a user file table (UFT) as part of his program that interfaces to the tables in the BIOS data structures. The user calls the I/O system by a series of executive (REX) services that all converge on a large central scheduling routine. This routine executes at the calling task level until an I/O operation is scheduled in the queue of the appropriate device controller. These services schedule such functions as READ, WRITE, REWIND, and WEOF.

Each task has a local logical file list called the task's File Assignment Table (FAT). It is located in the task's TCB. MAX IV has a similar list of logical files, called the Global File Assignment Table (GFAT), that is kept in the TCB of the standard Exceptional Condition (X) task. These tables are used by BIOS to find the assigned device when an I/O operation is requested for a particular logical file by a program. The logical files exist only as linked entries in these lists. The task's File Assign Table (FAT) is always used unless the desired file cannot be found in it. In this case, the Global File Assign Table (GFAT) is used. With this scheme, duplicate logical files may exist for different tasks, and yet unique device assignments can be maintained. Likewise, a single logical file name may be used for communications with a device by many different tasks.

Logical files may be assigned to other logical files, but the final assignment must be to a device (real, logical, or imaginary) that must be in the System Logical Device Table (LDT). Disk devices (not files) may be kept permanently in this table or may be added and deleted with the File Manager services, Job Control Directives, or special batch-processing utility programs. Each logical file has a name that the programmer must use when addressing the file. Each logical file entry has two assignments called the current assignment and the default assignment. With this feature, a task may change its file / device assignments at run time and restore the original assignments when desired. This is particularly useful for batch-processing tasks, where each new job can be given preset standard file assignments by using all the default assignments.

All I/O operations that cannot be started immediately are queued so that no program delay results due to a busy controller channel. Operation nodes in device controller queues are ordered by calling task priority and by chronological order for like task priorities. Devices that share a common controller channel have "slough" queues that prevent unmounted or unavailable devices from stalling the central controller channel queue.

When using BIOS directly, the user must expect to deal with the assigned device in terms of discrete physical records or blocks. Buffers must be supplied in the user's program that are capable of containing exactly one physical record. Each device has a different physical record size, therefore the user must often determine device size and adjust the program accordingly. The largest common buffer size for most standard devices does not exceed 128 words (1/2 memory page), although some special devices may be programmed with much larger buffers if the program is capable of using memory resources to such an extent. Data is transmitted to and from memory data buffers within the program body of the calling task. In a FORTRAN program, which uses subroutines to define access methods, these data buffers may be transparent to the user, but are allocated and used by the FORTRAN Run-Time Package, and reside in a labeled common area generated by the FORTRAN compiler or in an area near the end of the task body.

BIOS also supports a data chaining mode of operation for privileged programs. This mode performs an I/O operation that writes a continuous record on an output medium consisting of buffers gathered from non-contiguous memory areas. Data chaining is limited to mass storage devices (i.e. disk, magnetic tape).

BIOS assigns a data structure called an I/O node to the program requesting an I/O operation. I/O nodes are assigned as the program makes its I/O request. The assigned I/O nodes may come from a global I/O node pool or from a program's private I/O node list. If there are no nodes available in the global pool and the program has no free private nodes, then the program must wait for another task's I/O to terminate before a node is available. Private nodes are declared when a program is cataloged as a load module. There are tradeoffs between using private nodes and global nodes. A task's private I/O node(s) may be used only by the task owning them, so when the task is not executing, then the memory space used for the I/O node data structure is wasted. On the other hand the task may have to wait for another task's I/O operation to terminate if no global I/O nodes are available. The I/O nodes data structures reside in MAP 0 space which can be at a premium in large systems. In general a task should only use private I/O nodes if its I/O is extremely time critical. It is very rare that there are not global I/O nodes available.

The following are the data structures and linkage for a task's control block (TCB) and BIOS.

File Management System

There is a File Manager Executive (FMX) that organizes, maintains, and services multilevel files. FMX concerns itself with the definition and use of collections of physical records and the utilization of space on secondary storage volumes.

Secondary Storage Files

A disk pack (volume) is a basic subdivision of a disk controller. If the volume is removable, then the available secondary storage data base of the computer system extends beyond the number of physical disk transports available.

Any contiguous or noncontiguous subdivisions of a volume may be established as a named logical disc device. Such "devices" can be obtained in several ways. FMX calls these devices "real files". Any program can use these "devices" by assigning them to its logical files.

Fixed Partitions - Externally Described Files

At system generation time, one or more fixed partitions of a disk transport may be defined. These partitions are given unique device names and may be included as permanent entries in the System Device Table. Such devices can be associated with any logical file, and used directly by the programmer through the services of BIOS. These devices define contiguous single extents of disk medium that are fixed to the transport regardless of which pack is mounted. For this reason, the File Manager (FMX) calls them "externally described files". These devices may be addressed sequentially in a manner similar to a magnetic tape drive, or directly by a relative position index (16 or 32 bits). Each partition may be addressed independently of others. Such devices are sufficient for use with fixed-head disks and for parts of the System Residence Volume, but are not sufficiently flexible when many volumes can be alternately mounted on the same transport.

Real Files - Internally Described Files

Real files are defined by directories and other data structures written directly on the volume medium. Whether these volumes are mounted or removed, the definition of their files remains as part of the written medium. Secondary storage volumes that contain such files are said to contain "internally described files". When a volume is mounted, all the files described on it may be potentially opened by name as they are needed by programs. When such files are no longer needed, they are closed. When a volume is dismounted, all of its currently open files must be closed before the dismount is permitted. Disk files may be defined or expanded on contiguous or non-contiguous sections of disk media.

Utilities and Executive services permit the space on mounted disk volumes to be managed by batch-processing programs or by real-time tasks. New files may be added or expanded as needed on mounted volumes. Old files may be reduced in size or destroyed or renamed. Files that have become fragmented may be compressed (made contiguous).

Once a real file is opened for use by a program, it can be operated on with the services of BIOS. These services let the program deal with the file as a series of physical records (blocks or sectors). Both "block sequential" and "block direct" access methods are available within BIOS.

The File Manager also permits the simple definitions of disk partitions to exist on a volume under its control, which allows definitions to change with volume mounting.

Executive Services

Reentrant Executive Service subroutines (REX services) are available to all tasks. These services permit the following functions to be requested.

- File Management Services
- I/O Operation Queuing Services
- Task Execution Control Services
- Task Scheduling Services
- Byte-String Syntax Analysis Services
- Code Conversion Services
- Information Services
- Utility Services

A user can code reentrant and recursive REX services and add them to the system at system generation time.

Executive services may be invoked in many ways. The most direct way is to execute the desired REX instruction in Assembly language. REX services do not modify any registers other than those used for passing and returning arguments. REX services may be called directly from higher-level languages, such as FORTRAN using the standard subroutine conventions for the language. The system operator can execute many REX services that exert control over other tasks, by typing directives interpreted by the Operator Communication Task. The batch-processing programmer can execute many REX services by entering Job Control Directives (at the beginning of a job or between job steps). These services can only affect resources that have been committed to batch-processing tasks.

MAX IV supplies a special set of REX I/O services that are callable by high-level language programs for data acquisition and process control. Some of the things that these services provide for are: scanning, reading, and writing of groups of analog and digital inputs. The programmer does not need to be concerned with the addressing of individual points, but specifies only a group of points and the number of points they wish to access.

System Processors

MAX IV has a large number of system processors for use by batch-processing. It has the full complement of system processors (Editors, Link Editors, Library Update, Utilities, etc.) normally provided to allow for interactive batch-processing. It also provided a comprehensive set of file/data management processors.

Software Support

MAX IV supports the following high-level languages:

• FORTRAN IV	(standard)
• FORTRAN 66	(standard)
• FORTRAN 77	(optional)
• ANSI 74 COBOL	(optional)
• Pascal	(optional)
• C	(optional)

Batch-Processing

MAX IV provides software development facilities for both multiple batch streams and multiple interactive terminals. In other words the Modcomp Classic may also be used as a time-sharing multiprogramming system as well as a real-time system. The performance demanded for real-time computing allows the system to be configured as a powerful time-sharing system. In fact, most of the systems are configured to allow multi-user batch-processing in background while doing real-time computing in the foreground. Processes running in background are considered to be those below a certain task priority level (mutually agreed upon depending on application). Foreground tasks are those running above this agreed upon priority level. One thing to remember is that the foreground real-time tasks can starve the background ones, but that is the price you pay in a real-time system.

MAX IV provides a powerful command line interpreter called Job Control. The batch-process is monitored by a job control task that interprets job control statements supplied serially by a command input file. Typically the command input file is assigned to the users terminal but it may be assigned anywhere that the user desires. The job control language is a flexible procedural language that allows the user to control the execution and system resources of system processors (such as the FORTRAN compiler or any user written program). Job control procedures may be stored on some file (usually on disk, but could be magnetic tape or other), and may be executed by a single user defined job control macro statement.

MAX IV also provides a Terminal Monitor Program (TMP) for monitoring multiple terminals. TMP also provides a comprehensive set of directives that allow the batch-processing programmer to interface with the REX services that affect the local batch-process.

A Batch Task is provided by MAX IV for doing batch-processing. This Batch Task may be customized with whatever resources and options that the user wants to do batch-processing. The Batch Task may be duplicated and assigned to interactive terminals to allow multi-user interactive batch-processing. The number of users allowed is limited generally by the number of terminals connected to the system. Typically all of the interactive Batch Tasks will be assigned to the same task priority level (hopefully a low priority, typically 250 - 255). The TASKMASTER's scheduling algorithm may be set to round robin task scheduling for tasks on the same priority level. What we have now is a time-sharing system for all of the interactive Batch-Processing users. In fact it is a very good time-sharing system (performance wise) until some real-time task comes along and hogs the system.

Operator Communication

MAX IV provides for an on-line man/machine dialogue between the operator and MAX IV. This is done through the Operator Communications (OC) task. The operator virtually has complete control of the system. The operator can assign devices to logical files, define logical files, assign priority levels to tasks, control the execution of tasks, obtain system or task information, etc., by typing in OC commands. The OC task may be accessed from any terminal connected to the system using the Terminal Monitor Program (TMP). There is a password protection scheme (though rarely used) that restricts unauthorized access to OC. The OC task typically has the highest task priority level in the system.

System Integrity Protection

There is much less protection of the system (both MAX IV and user processes) than typically found on a time-sharing system. The hardware and software mechanisms for protecting the system are available. It is just that the flexibility that the user has can often override some or all of the protection. For example a user program may be cataloged as being privileged which would allow it to execute all of the privileged CPU instructions normally used only by MAX IV. One user can affect the execution of another or the user can adjust their own task's priority level. There are many other examples, but I am sure by now, that you realize that a user can literally destroy the system. However this flexibility given to the users is very necessary for a system to do real-time computing. The hardware and MAX IV provide the mechanism for system integrity protection; however the application programs and system configuration must ensure that the protection may be employed as much as possible. In other words, MAX IV provides for system integrity protection, it is up to the systems application to use it.

MAX IV and the hardware provide memory protection on a 512 byte page basis. Protection is achieved by access rights that are associated with each virtual page. These access rights determine the degree of access for each page that a particular task is granted. Access rights control access to memory by all of the system elements including I/O. These access rights are:

AR = 11 - Read, write, and execute
 10 - Read and execute
 01 - Read operand only
 00 - No access

The task's page addressing space (virtual page) is protected and not the actual memory page. This allows tasks that access the same actual page (shared page) to have different access rights for the same actual page.

When an access rights violation occurs, the hardware sets the hardware interrupt level 2 (System Protect Violation) and MAX IV will abort the guilty task with a System Protect abort.

Another form of protection is offered by the CPU's privileged and non-privileged modes of operation. Any non-privileged task that attempts to execute a privileged instruction will cause a System Protect Violation with the same results as a memory access violation.

MAX IV provides for protection of unauthorized interference of tasks with each other. All tasks have an influence level as well as a task priority level, that together can protect a task from unauthorized interference from another task. A task, whose influence level is lower than the influence level of another task, may not affect the execution of that task. A task may not set its own priority level higher than its influence level.

New Real-Time Systems

As stated at the beginning of this paper, traditionally real-time computing has been a realm of proprietary operating systems with real-time applications written in FORTRAN and assembly language. In the past, these systems adequately served the needs of the real-time world. Many of these real-time systems developed 15 years ago are still being used today. However the real-time world is now changing, demanding new systems to be developed.

Each real-time application generally required new software to be developed. Many times, new software was required in order to upgrade to a vendors new hardware platform, as software compatibility was not maintained. Modcomp systems have maintained complete software compatibility among all of their 16-bit computers for the last 20 years. This compatibility was extended to their earlier 32-bit computers which gave the user the option of running in 16-bit mode. However, that is no longer the case with the newer Modcomp 32-bit machines as maintaining compatibility with the older systems carries too high a price.

The escalating costs of software development has created a need for real-time applications programs to be portable to other machines without massive conversion efforts. Portability is required whether moving software to other similar real-time applications or upgrading to new state-of-the-art hardware platforms. The high software development costs have also created a need for the ability to develop new software in a programming language that is widely used. Developing software in assembly language is often very time consuming, expensive, and offers almost no portability.

The real-time systems vendors have been forced to look for an operating system that will reduce software development costs, and satisfy real-time computing needs.

UNIX operating systems are designed for portability, and generally offer the high-level "C" language as their native language. UNIX and the "C" language are widely used on many systems today. Therefore the UNIX operating system would be a likely candidate for developing applications software. The only problem is that the UNIX operating systems are basically time-sharing systems, and are not suited for doing real-time computing. However some real-time vendors such as Masscomp, Harris, and Modcomp have set out to develop real-time UNIX operating systems based on AT&T's UNIX V operating system. The major weaknesses of UNIX System V for real-time applications are the lack of a preemptive kernel and the limited ability to preallocate resources for a process. A preemptive kernel is one in which a high-priority process can preempt the CPU from an executing process at a lower priority. Without this feature, a high-priority process can be blocked by a low-priority process. UNIX System V has a non-preemptive kernel: an executing process relinquishes the CPU to wait for some hardware or software event, or when it receives an interrupt from the hardware, but not because it is blocking a high-priority process.

Modcomp has successfully developed a high-performance Real-Time UNIX operating system, that satisfies both the UNIX and real-time worlds. Their Real-Time UNIX operating system called REAL/IX is described in the rest of this paper. The next section will deal with standardization before I get into the functional description of the REAL/IX operating system.

Standardization

Over the years, a number of variations of UNIX operating systems have evolved. While all these variants have a common ancestor, the differences between versions was, ironically, starting to threaten the portability that is one of the major advantages of the UNIX system. A number of attempts have been made to standardize the user level interfaces of the operating system. The two major ones are AT&T's System V Interface Definition (SVID) and the ANSI/IEEE committee 1003 (POSIX).

The AT&T SVID includes a Base Interface Definition plus a series of extensions. The Base Interface Definition describes the services that all System V operating systems must provide. The extensions to the base are not required to be present but, when a component is present it must conform to the specified functionality. The first volume of SVID was published in the spring of 1985; since then two more volumes have been published and more are under development. The REAL/IX operating system conforms to SVID and passes the System V Verification Suite (SVVS).

The POSIX group is a standardization effort that includes a broader base of corporations. They began meeting at about the same time the SVID was published, and have now released a draft of the IEEE POSIX Standard 1003.1. With the growing interest in real-time operating systems, a subgroup was spun off to standardize a set of user-level extensions for real-time operating systems based on the POSIX standard. At the middle of 1989, the IEEE POSIX Standard 1003.4 for real-time operating systems was under development. The REAL/IX operating system includes facilities that satisfy the topics identified by the standards group. Modcomp is an active participant in the POSIX 1003.4 committee and intends to support the standard when it is published.

REAL/IX kernel modifications have been made in such a way that SVID-compatible application code will run on the REAL/IX operating system without modification, although modifications may be required to realize full real-time performance. The REAL/IX operating system maintains full binary compatibility with Motorola System V/68, Release 3 Version V for user-level programs.

REAL/IX Operating System

The complete description of the REAL/IX operating system is far beyond the scope of this paper. What is intended is to cover some of the REAL/IX extensions to the UNIX System V operating system that provides for real-time computing. The first version of the Modcomp REAL/IX operating system (REAL/IX.1) was actually a hybrid operating system with the UNIX System V kernel for UNIX type applications layered with the Modcomp MAX 32 kernel for real-time applications. This system was designed to run on the Modcomp medium-performance TRI-D 92xx series 32-bit hardware platforms. Since that time Modcomp has developed the new REAL/IX.2 operating system designed to run on their new TRI-D 97xx high-performance 32-bit hardware platforms. REAL/IX.2 is not a hybrid operating system, but is a true high-performance real-time UNIX operating system. The new REAL/IX.2 operating system is what will be described in this paper.

Comprehensive Beta testing of the new REAL/IX.2 operating system and TRI-D 9750 hardware platform has been completed. REAL/IX.2 achieved higher real-time performance than the MAX 32 real-time operating system.

The REAL/IX.2 operating system is a fully preemptive, low latency, real-time UNIX kernel that provides a user environment similar to that of UNIX System V but with extensions that provide the capabilities required by real-time applications. These are summarized here by the major subsystems of UNIX operating systems: process subsystem, interprocess communications, file subsystem, and I/O subsystem.

Process Subsystem

The REAL/IX process subsystem includes the following enhancements required by real-time applications:

- Fully preemptive kernel with synchronization mechanisms to ensure data structure integrity.
- Fixed-priority based process scheduler for critical real-time processes.
- Real-time timer mechanisms.
- Enhanced memory management facilities that support preallocation of memory resources.

Interprocess Communications

UNIX System V supports several interprocess communication mechanisms: signals, shared memory, semaphores, and messages. These facilities are fully supported on the REAL/IX operating system, with the following extensions:

- Common event notification mechanism that is more reliable than the signal mechanism (since more than one signal of a given type can be queued at once), and allows signals to be received synchronously as well as asynchronously.
- Binary semaphore mechanism, which is a faster version of the UNIX System V semaphore mechanism.

File Subsystem

The REAL/IX file subsystem provides the following enhancements to the UNIX System V file subsystem:

- Alternative file system architecture that supports preallocation of space and contiguous files.
- Ability to bypass the buffer cache for file I/O.
- Synchronous updates of physical storage devices.
- Larger logical block sizes for faster throughput to files.
- Improved locality of reference.

I/O Subsystem

The REAL/IX I/O subsystem enhances the UNIX System V I/O subsystem with the following capabilities required for real-time computing:

- Support for asynchronous I/O operations.
- Disk I/O queuing prioritized according to the process priority.
- Direct I/O between a user-level program and a device.
- Connected interrupts for notifying a user-level process of a hardware interrupt.

Other Enhancements

A number of other modifications have been made to the REAL/IX operating system to provide the control and performance required for real-time computing. Most of these are internal modifications that are completely transparent to the users, but some will be visible:

- A number of daemons have been added to the system to provide necessary kernel functionality (such as writing error messages to the console) without degrading system interrupt latency. Daemon processes run at user-level but do system-wide functions. They are either kernel-resident or spawned by the init process and exist through the lifetime of the system. Many of these daemons and other system processes have tunable parameters that determine their execution priority.
- The SCSI interface simplifies the administrative tasks involved in adding disk and tape devices to the system as well as providing the flexibility of the SCSI interface.
- System calls can be installed into the kernel by entering them in a table and relinking the kernel. On other UNIX operating systems, system calls (which execute with less overhead than library routines) can only be added by customers who have source code and can recompile the entire kernel.

The following table lists the commands, system calls, library routines, and data structures that provide the functionality described above:

Summary of REAL/IX Features

FEATURE	ENTITY	DESCRIPTION	ENTITY	DESCRIPTION
Realtime Priorities and Privileges	setrtusers	assign realtime privileges to a user	setrt	set realtime privileges for a process
	clrrt	remove realtime privileges for a process	getpri	get scheduling priority
	setpri	set scheduling priority	suspend	suspend calling process
	resume	resume a suspended process	switch	switch into a process
	relInqulsh	voluntarily give up CPU	setslice	set CPU time slice size
Kernel Synchronization	psema	decrement value of kernel semaphore	vsema	increment value of kernel semaphore
	cpsema	check value of kernel semaphore	cvsema	set spin lock
	valusema	release spin lock	spsema	check value of spin lock
	svsema		valuelock	
Realtime Timer Mechanisms	absInterva IncInterval	set value of process interval timer	getInterval	get current value of process interval timer
	gettimer	return current system time	gettimerid	get a process interval timer
	reIntervId	release a process interval timer	resabs resinc	return resolution and maximum value of process interval timer
	restimer	get resolution and maximum value of system wide realtime timer	settimer	set current value for a system-wide realtime timer
	Itimerstruc	format of process interval timer value	timestruc	format of system-wide realtime timers
Memory Allocation	resident	lock memory segments in memory	stkexp	preallocate memory for stack
Common Event Mechanism	evctl	event control operations	evget	get an event identifier
	evpost send.event	post an event to a process	evrcv evrcvl	receive a queued event synchronously
	evrel	release an event identifier		
Binary Semaphore Mechanism	bsget	get a binary semaphore	bsfree	free a binary semaphore
	bslk	lock a binary semaphore	bslkc	conditionally lock a binary semaphore
	bsunik	unlock a binary semaphore		

FEATURE	ENTITY	DESCRIPTION	ENTITY	DESCRIPTION
Fast File Systems	estat efstat	get status of extended file	trunc	truncate file space
	prealloc	preallocate contiguous file space		
Asynchronous I/O	arinit awinit	Initialize structures for asynchronous I/O	aread awrite	Initiate asynchronous I/O transfer
	acancel comp_cancel alo	cancel asynchronous I/O operation	arwfree	free structures for asynchronous I/O
	alo	driver entry point for asynchronous I/O	comp_alo	mark completion of asynchronous I/O transfer
Fast File Systems	estat efstat	get status of extended file	trunc	truncate file space
	prealloc	preallocate contiguous file space		
	arinit awinit	Initialize structures for asynchronous I/O	aread awrite	Initiate asynchronous I/O transfer
Asynchronous I/O	acancel comp_cancel alo	cancel asynchronous I/O operation	arwfree	free structures for asynchronous I/O
	alo	driver entry point for asynchronous I/O	comp_alo	mark completion of asynchronous I/O transfer

Several standard components have enhanced functionality to support these features.

System Concepts

The REAL/IX operating system can be viewed as a layered entity, as illustrated below. Users always interact directly with the user-level, usually through the shell. From the shell, one may call utilities and user programs, which in turn may call library routines. In the user-level of the operating system, one opens and closes files (which may be actual data files, or may be special files that correspond to system devices), reads and writes data, and so forth.

The kernel-level of the operating system is accessed through system calls, which may be called directly by utility programs and user processes, or may be called through library routines. Kernel-level processes (or functions) handle the details of interacting with the internal structures of files, processes, and devices. The hardware-control-level of the system is accessed through kernel-level processes, usually drivers

UNIX Operating System

Allocating Memory

REAL/IX memory allocation is similar to that on other UNIX operating systems, with extensions to provide explicit control over memory allocation in critical real-time application programs. These extensions provide the real-time programmer with complete control over REAL/IX's demand-paging subsystem. To guarantee response time, users are allowed to pre-page and lock all pages (instructions, data, shared memory, and stack) of a program into memory. At the programmer's option, the operating system will notify a real-time process of any attempts to grow the stack or data portions of the process's data segment.

The underlying philosophy of memory allocation is different for real-time and time-sharing processes. The time-sharing philosophy is to avoid consuming any more memory than is absolutely necessary so that all processes have equal access to memory resources. For critical real-time processes, the emphasis is on providing optimal performance for the program. Consequently, real-time programs typically preallocate a generous amount of memory and lock all resources they might need into memory. The REAL/IX operating system can be used to implement either philosophy.

The REAL/IX operating system uses demand paging, so the operating system does not allocate any memory for a process when it is initialized. Rather, the process is allowed to start executing at its entry point, which causes the process to page fault text and data pages as they are referenced. As the data segment (which is composed of the stack and data regions) outgrows that memory, the system allocates more physical pages. This scheme conserves memory and is appropriate for many applications, but the overhead incurred is unacceptable for critical real-time programs.

Most UNIX operating systems provide the `brk` and `sbrk` system calls to preallocate virtual space for the data region. The end of the data segment is called the break. `brk` and `sbrk` allow you to specify a new location for the break, with `brk` specifying an absolute address and `sbrk` specifying an address relative to the current break.

The REAL/IX operating system also provides the `stkexp` system call, to preallocate virtual space for the stack. You can specify either the absolute size of the stack or the increment by which the stack is to grow. For programs that need memory allocated for data space, the `malloc` mechanisms are a simple, general-purpose memory allocation package. Real-time programs should call `malloc` only during the initialization part of the program, or use `brk` or `sbrk` to preallocate data space before calling `malloc`. Note that one program should not call both `malloc` and `brk/sbrk`.

Paging enables the operating system to provide good performance to a number of programs executing at the same time. However, the overhead associated with accessing processes or data that have been paged out is significantly more than the overhead involved in accessing processes or data that are resident in memory.

The `plock` system call allows you to lock text and data segments into memory. The `shmctl SHM_LOCK` system call allows you to lock shared memory segments into memory. These calls lock segments into memory when they are first accessed. They can be used with the system calls that preallocate memory, or can allow the operating system to allocate memory as needed.

Critical real-time processes can lock segments into memory during process initialization with the "resident" system call, so that the first attempt to access a segment does not incur the overhead of loading it into memory. The "resident" call requires that memory be preallocated as discussed above. For critical real-time processes that preallocate memory, expanding memory beyond the preallocated limits is usually considered a fault, although the action taken for such a fault is at the discretion of the programmer. If desired, the "resident" call will post an event to the process if the memory allocated is inadequate for the stack or data region.

Scheduling

Scheduling determines how the CPU is allocated to executing processes. Each executing process has a priority that determines its position on the run queue. On the REAL/IX operating system, the run queue consists of 256 priority "buckets", divided into two major parts as illustrated below. Processes executing at priorities 128 through 255 (time-slice priorities) utilize a time-sharing scheduler implemented in the "onesec" process. The "onesec" process replaces UNIX System V's "sched" process and is always Process 0. It runs roughly once a second and provides many housekeeping functions (including priority recomputation) that are traditionally performed by the clock interrupt handler. The priority of "onesec" is controlled by a tunable parameter. Processes executing at priorities 0 through 127 (real-time priorities) utilize a process priority scheduler implemented internally. Each of these schedulers utilize different scheduling algorithms and are discussed below.

REAL/IX Process Priorities

Priority Buckets	Description
0	Real-time priorities are changed only by explicit request of program or administrator.
1	
.	
.	
.	
126	
127	
128	Time-sharing priorities are dynamically adjusted by the operating system.
129	
.	
.	
.	
252	
253	
254	Non-migrating priorities execute only when all other processes are idle.
255	

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 - 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 (by default, the time slice is 11 years, but this can be changed with the "setslice" system call).
- Because the REAL/IX operating system has a preemptive kernel, 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 in which processes are executed. These parameters are determined differently for time-sharing and fixed-priority scheduled processes.

Time-Sharing Scheduling

Processes executing at priorities 128 through 255 utilize a time-sharing scheduler similar to that on other UNIX operating systems. The operating system varies the priorities of executing processes according to a specific algorithm. For instance, interactive processes gravitate towards higher priorities (since their actual run times are relatively small compared to their wait times), and processes that recently consumed a large amount of the CPU are relegated to lower priorities. A process's only control over its priority is with the "nice" system call and command, which lowers the relative priority of a process or, if issued by the superuser, can grant the process a more favorable priority.

The kernel allocates the CPU to a process for a time quantum, which is determined by the MAXSLICE tunable kernel parameter. If the process has not released the CPU at the end of that time quantum, it is preempted and fed back into the queue at the same priority. When the kernel again allocates the CPU to the process, the process resumes execution from the point where it was suspended. Once a second, those time-sharing processes that have consistently used their whole quantum are shuffled to lower priorities.

This time-slice scheduler has the advantage of equitably distributing the CPU among all executing processes. It is not adequate for critical real-time processes, which need to execute in a determinate (preferably fast) time frame. For this reason, the REAL/IX system supplements the traditional time-sharing scheduler with the fixed-priority scheduler.

Fixed-Priority Scheduling

Processes executing at priorities 0 through 127 utilize a fixed-priority scheduler. A process establishes its own priority with the "setpri" system call. The operating system will never automatically change the priority of a process executing at a real-time priority, except during a boost operation; only the process itself or a user with real-time or superuser privileges can change the priority. A process will execute only if there are no processes with higher priorities that are runnable at this time.

Runnable processes at the same priority are arranged in a circular, double-linked list. A round-robin scheduling scheme is used for processes at the same priority, with the ability for a process to relinquish its time slice to another process at the same priority. A process can also use the "setslice" system call to establish quantum; the default is 11 years.

The fixed-priority scheduler allows a critical real-time process to "hog" the CPU as long as necessary to finish. If processes at high real-time priorities are using the CPU, low-priority real-time processes and time-share processes may never get to execute. Note that a process scheduled at priority 100 will run just as fast as a process scheduled at priority 1, if no processes are scheduled at a higher priority. Because a high-priority "runaway" process may never surrender the CPU, it is recommended that the console or a user terminal be set to run at priority 0 or 1, and no other processes (other than critical system processes) run at that priority, to ensure that you can always regain control of the system.

Locks to Preserve Data Structure Integrity

The REAL/IX operating system uses spin locks and suspend locks (or kernel semaphores) to ensure data structure integrity in the preemptive kernel. If two processes access the same global data structure, it is important that the first process completes any update to that structure before the second process accesses it. In other UNIX kernels, this is handled by disabling interrupts, to prevent an interrupt handler from accessing a data structure that was being manipulated by process-level kernel code. If the preemptive kernel were implemented without locks, a higher-priority process could cause a context switch from a lower-priority process even though it is in the process of updating a data structure, and thus corrupt the structure.

In a non-preemptive uniprocessor environment, data structure integrity can be preserved by manipulating processor execution levels to prevent interrupts when updating a structure. However, this is inadequate for a multiprocessor configuration because the interrupt handler, or another process may be executing on a different processor than the process-level routines. The locking mechanisms could enable REAL/IX to run on a multiprocessor configuration, where all processors operate on a symmetrical, peer-to-peer basis, and each processor can simultaneously execute user-level code, process-level kernel code, and interrupt-level kernel code.

Synchronization in Compatibility Mode

Other UNIX operating systems provide kernel-level synchronization with the "sleep/wakeup" functions to block and unblock a process, and the "spl" (set priority level) function to disable interrupts. The REAL/IX system provides three compatibility modes that allow drivers to be ported from UNIX System V without rewriting the synchronization facilities. The three modes are:

- Non-preemptible - kernel preemption is turned off when the process is running.
- Major-device semaphoring - one semaphore is set for the major device (that is, the driver itself). This is implemented in the switch table, so that only one instance of the driver entry point can execute at a time.
- Minor-device semaphoring - one semaphore is set for each minor device (that is for the actual device controlled by the driver).

Drivers installed using these compatibility modes, may not realize the full performance enhancements provided by rewriting the drivers to use kernel semaphores and spin locks, but should perform similarly to how they do on UNIX System V.

Timers

The operating system uses timers to schedule "housekeeping" operations that must be run periodically, and to provide a mechanism through which real-time processes can schedule events.

The REAL/IX clock interrupt handler differs from that on most other UNIX operating systems, in that the interrupt blocking time has been bounded to never block interrupts longer than 100 microseconds. To ensure this real-time constraint, some of the traditional interrupt handler functions have been moved to high-priority processes (daemons), that are triggered from the clock interrupt with "vsema" or "cvsema: operation.

Real-time Timers

Real-time processes need tight control over timing constraints in order to schedule specific events to occur in the system. These timers must be flexible, providing the real-time application with the ability to use either absolute or relative time. They must also provide the ability to schedule events in increments of seconds (like other UNIX timer functions) or microseconds for critical real-time applications.

The REAL/IX operating system provides a set of timers to support the requirements of real-time processes for scheduling events within the system. The REAL/IX timer implementation provides two types of timer mechanisms which real-time applications can access through system calls:

- System-wide real-time timers provide the application with the ability to read, write, and get the resolution of a system timer. This type of timer corresponds to the system time-of-day clock and represents the current time in seconds and nanoseconds since January 1, 1970. This timer is ascending in nature and is updated by the system at the frequency of the system clock. This system timer provides access to the system time in a unique format (seconds and nanoseconds), and is used to provide the process interval timers with a method of obtaining the current system time in a readily usable format.
- Process interval timers are designed to be used by one or more real-time processes to schedule system events within a very fine time scale, from a few seconds down to a 1/256 second. The interval timers can be set by a real-time process to expire based on a time value that is relative to the current system time, or a time value that represents an absolute time in the future. These timers can be set to be "one-shot" or periodic.

A list of free process interval timers is defined at system generation (SYSGEN) time. The interval timers from this list may be allocated by real-time processes during process initialization or during normal execution. By using SYSGEN parameters, the timer mechanisms can be customized for a particular application that requires varying amounts of timers to be available for process allocation.

Privileges

The REAL/IX operating system recognizes three basic privileges: regular, superuser, and realtime). Regular privileges are the standard privileges granted to a user. Superuser privileges are the traditional administrative privileges, which allow access to all files and processes on the system; a number of administrative functions can be executed only by the superuser. Realtime privileges can be assigned to a user or a process and are required to use a number of the special REAL/IX features for real-time programming; restricted assignment of these privileges ensures that a time-sharing program cannot inadvertently affect the real-time responsiveness of the system.

Real-time users and processes have more privileges than regular users, but fewer privileges than the superuser. A superuser can do anything that a real-time user can.

Utilities

The REAL/IX operating system includes the standard UNIX System V utilities.

Acknowledgements

This work was sponsored by the U.S. Department of Energy under Contract No. DE-AC03-89ER51114.

END

DATE FILMED

11 / 29 / 90