The Design of a Reduced Function Operating System

James R. Bergsten
Founder, President and Chief Technical Officer
Ark Research Corporation

Introduction

This paper describes the design and implementation of a *Reduced Function* Operating System. Conceptually similar to a *Reduced Instruction Set Computer* (*RISC*), our goal was to create a *real time operating system* (*RTOS*) that is efficient, easy to maintain, portable, and extendable.

Virtually all of the concepts implemented in this RTOS can be traced to work that has gone before and will no doubt be familiar to the reader. Nevertheless, there was neither a need nor an attempt to be compatible with any previous or existing operating system.

I will describe the design decisions resulting in the development of RTOS features and functions, and review the development history, resources, and philosophies used to develop and test this system.

I believe that this effort has enabled us to successfully leverage many of the best concepts and practices of operating system design into a real world product.

Table of Contents

Overview

The RTOS was created to support the development of a high performance, functionally rich advanced storage controller (*ASC*). This storage controller is described in *The Design of an Advanced Storage Controller* document, by this author.

One goal was to create this storage product using commonly available, *off the shelf* hardware components, avoiding the cost and risk of developing proprietary hardware. The RTOS and application were also developed to be independent of, and therefore easily portable to existing or new hardware platforms. At the start of the project (1995) no generally available RTOS seemed to meet these requirements.

A very small team developed the RTOS and ASC, tested, and released the ASC product family in slightly less than four wall clock years.

About ninety percent of the RTOS and ASC code is written in *C* (**not** *C++*). The remaining code is written in assembler. The decision to develop **any** assembler is based on two criteria:

1. If a routine is hardware *platform* dependent, it was written in assembler. This gives developers a clear indication of routines that need to be modified when porting the RTOS to a different hardware platform.

   System initialization, first-level interrupt handling, and task switching glue logic routines are written in assembler. As our I/O devices are all PCI or ISA bus based, and are thus largely platform independent, device drivers were written in *C*.

2. If a routine could be designed to run **significantly** faster if written in assembler, it was. For example, routines to generate and check TCP/IP checksums, data encrypting, and data compression are good candidates for assembler code.

Objectives

Here are some of the design objectives for this operating system:

- Minimize system overhead to maximize application performance.

- Provide excellent response times, as most applications handle time-sensitive real-time transactions.

- Create the minimum set of RTOS function necessary to support applications -- a small but complete set of system support routines to simplify application development and shorten new developer learning curve.

- Provide complete recovery or minimal disruption from software or hardware failures.

- Separate functions – for example, device dependent code only resides in a device's driver, conversely, any driver support code that could be shared is placed into common routines

- Protocol independence – for example, physical layer support is separate from logical layer support.

- Features traditionally added to operating systems in a major redesign or rewrite were incorporated into the initial design, to avoid the higher cost of subsequent redesign, development, and debugging. These included:

  Multiprocessing support.

  Hardware platform independence. All RTOS code and data structures provide conditional compilation to support both *Little Endian* versus *Big Endian* architectures (the byte-order multi-byte values are saved in memory), and, when necessary, data transferred between systems or saved externally are always converted into a platform-independent format.

- No run-time library dependencies. Except for a very small set of intrinsic functions (routines such as *memcpy*) that generate in-line code), RTOS routines do not call any standard C library routines. While this adds to the developer learning curve, it eliminates additional porting, support, and cost issues, such as unavailable runtime library source code, subtle library incompatibility, reliance on outside vendor problem resolution, and library licensing issues.

System Functions and Features

This section of this paper describes the major features and application support functions provided by RTOS. To help illustrate, a sampling of relevant RTOS calls, shown as C function prototypes are often provided.

Initialization, Shutdown, and Restart

System initialization comprises the code required to get from the first instruction loaded by the hardware boot sequence to the point where the system is ready to process work. Conversely, *shutdown* and/or *restart* must terminate the RTOS in an orderly fashion, saving system and application states where necessary.

Most of an operating system's hardware platform-dependent code resides in these areas. As the first hardware platform supported by RTOS was Intel *x86* architecture, some insights into that particular environment are included in this section.

Loading the RTOS from Flash or Disk

Every hardware platform must have the capability to load a small amount of code and data into a known memory location and start execution of instructions from a known state. On the Intel platform, ROM-based code (called the *BIOS*) performs perfunctory hardware initialization, testing, and initial program load (*boot* – from the word *bootstrap*). A very small amount of code and data (512 bytes) are loaded into a fixed location in memory, then instruction processing begins at the first location in that memory. This code is usually referred to as a *bootstrap loader*, as its function is to load the rest of the RTOS into memory.

It is possible to design hardware cards containing *on-board* code and data, located within a reserved memory range. By scanning for a *signature* (a known string at known locations), the BIOS can execute these instructions as part of the boot process -- this feature is used by hardware manufacturers to place device specific hardware initialization code on their plug-in cards. To allow the RTOS to boot without the need for floppy disks or dedicated hard drives, we developed a small hardware card containing *flash memory* to contain several versions of our RTOS and user configuration files.

Three different versions of bootstrap code were created to load the RTOS. The first loads the code from a floppy drive, the second from a hard disk, and the third from flash memory. In all three cases, the code performs the following functions:

1. Load the remainder of the boot loader if necessary.

2. Load the RTOS code and data. To allow for software upgrades, new version back off, or recovery from flash memory failure, the bootstrap loader can load from a number of potential locations or *slots*. The user can choose which saved RTOS image to load.

3. Load the current configuration file contents. Like the RTOS, there are several configuration slots to choose from.

4. Setup to pass system configuration information to the RTOS, such as memory size, display type, etc.

5. Transfer control to the RTOS initialization code.

Transfer from Real to Protected Mode

For backward compatibility, Intel architecture supports two *modes* of operation. The older one is called *Real Mode*. This is a very limited mode -- compatible with the very first Intel processor chips (this is also sometimes referred to as *16-bit mode*). This mode is very restrictive in terms of function and memory size (for example, only one megabyte of memory is addressable). You don't even want to know about Real Mode *segmented addressing*.

Protected Mode supports the full feature set of the architecture.

There were several development challenges using Protected Mode:

1. For all practical purposes, the BIOS run-time functions are not accessible in Protected Mode. This required development of functions that might otherwise might have been available via BIOS calls. However, not relying on the BIOS makes the RTOS much more portable.

2. The transition to and from Protected Mode was poorly and inaccurately documented, and turned out to have many order dependent steps. Any incorrect sequence resulted in either locked up or reset hardware. The correct sequence was arrived at by trial and error—years later Intel documented the procedure correctly.

3. The development tools did not allow a mix of 16-bit and 32-bit code. This did not turn out to be an issue, as we coded the bootstrap loaders to run in 16-bit Real Mode, do the mode change, and transfer to the all-Protected Mode 32-bit RTOS initialization code. One interesting challenge was to load the RTOS above the one-megabyte Real Mode limit to avoid any RTOS code-size limitations. This relocation was performed using a BIOS function normally used to extend the memory range of some MS/DOS programs.

Configuration and RTOS Image Files

One design requirement of the system was to give users the ability to completely configure the RTOS and its applications. Since the user's configuration information must be available to the system and its applications at all times, even during the most basic system initialization, the bootstrap loader(s) are responsible for loading the configuration data. The system runs using default values if no configuration file exists.

The configuration file (and each RTOS command) contains one or more character string statements of the form:

```
    verb <<parameter1 <values>> <parameter2 <values> ...>
```

where *verb* is the action to be performed, followed by optional parameters and parameter values.

For example, this command:

```
    timezone -8 0 pst pdt
```

sets the time zone of the system to Pacific Standard Time and enables Daylight Savings Time.

There is no required *order* to the statements in a configuration file. The file is always accessible to RTOS and applications -- each can access statements whenever necessary (although many parsed statements generate internal data structures that are easier to access then the raw configuration file statements). The configuration file is saved in *source code* ASCII text format – there is no configuration file compiler.

RTOS supports the ability to load a configuration file from a number of places:

1. From a floppy disk containing the RTOS (for a self-contained bootstrap and system set).

2. From a hard disk file.

3. From one of a set of flash memory slots.

One flash memory or hard disk block contains a directory of system and configuration slots, defining the currently active and backup system and configuration images to load.

It is possible to load, change, and delete system load images and configuration files via standalone utilities using floppy disk data, via File Transfer Protocol (FTP) uploads and downloads, and via user commands while the system is running.

In the event of a configuration or system image file error or failure, it is possible to change to another slot so that one cannot end up with an unbootable system..

Paging, Caching Parameters, and other Hardware Dependent Tables

RTOS runs in what Intel architecture calls the *flat address space* model (the RTOS **does** use Intel segment registers to prevent attempts to execute code outside of the code region, prevent access to non-existent memory, and detect stack underflow).

The RTOS does not dynamically alter the translation of memory address from virtual (or segmented) memory to real memory. Since all virtual addresses are equal to real addresses, I/O operations to contiguous real memory do not incur the overhead of real-time generation of mapped I/O lists (sometimes referred to as scatter-gather lists, or SG lists). As all code and data permanently reside in memory, there is no need for memory paging.

Although memory is mapped *virtual equals real* (*V=R*), the system runs with address translation hardware **enabled**. This is done for two reasons:

1. Protection – undefined, unallocated, and private memory are marked *invalid* in the page tables, preventing inadvertent access. Pages containing code are marked *read and execute only*, pages containing read/only data are marked *read/only*.

2. Caching Parameters – many hardware platforms use address translation hardware to define individual page caching parameters (such as whether pages can be cached in processor cache, and whether modified in-cache data are immediately written back to memory).

Page directory (segment) and page tables are dynamically created, based on real and I/O memory space size. Other hardware platform tables are also created, notably hardware and software interrupt vector tables (in the case of Intel architecture, IDT, GDT, TSS). Stack space for each CPU is also allocated, as well as system event trace tables, instruction profiling tables, etc.

The RTOS uses 4,096-byte memory pages, each aligned to a 4KB boundary.

Device and storage checking

By using a set of fixed and *walking* bit patterns, all installed main storage are validated. The RTOS supports, and when available, requires that memory *Error Checking and Correcting* (ECC) is enabled. The RTOS code and read/only data are dynamically *checksummed* – this checksum is used to insure that the code was not corrupted in the case of a system failure, recovery, or restart. RTOS ensures that all hardware error recovery is enabled, for example, bus parity, machine check reporting, bus error reporting, etc.

All configured I/O busses and devices are scanned and initialized .More details on this can be found in the *Device Driver* section below.

Initial-Task Creation

Once the system and storage are initialized, initial and permanent tasks are created (but not started). These tasks:

1. Run once to perform application or additional system initialization. For example, tasks can be created to setup passive TCP opens for Telnet, FTP, or HTTP services.

2. Provide ongoing system monitoring, control, statistical gathering, or repeated function. For example, some permanent tasks might include: Date/Time display update, Front panel update, storage and/or device *scrubbing* (checking), ARP message handling, keyboard handling, system message logging and queuing, and so on.

Multiprocessor (MP) setup and startup

Before the system begins to process tasks, additional central processing units are initialized to run the task dispatcher.

As memory and most system tables are already setup, for each installed, on-line processor, multiprocessing startup involves:

1. Allocating resources unique to each processor, such as a unique *kernel mode* system stack and a processor-specific data structure for control and statistics gathering

2. Creating multiprocessor bootstrap code to allow the processor to load appropriate internal registers, switch to the appropriate running mode (in the case of Intel, Protected Mode), enable the receipt of appropriate interrupts, and branch to the task dispatcher.

Once all other processors are started, the *boot processor* branches to the task dispatcher and begins to handle runable tasks.

Shutdown, Warmboot, Coldboot, and Restart Commands

RTOS commands are provided to shut down and/or restart the system. The four commands are:

`SHUTDOWN`	Orderly shutdown of the system. Any of the following three commands can be entered following a shutdown.

`RESTART`	Orderly shutdown, followed by a restart of the currently resident code and configuration file. This is the fastest way to restart a system.
`WARMBOOT`	Orderly shutdown, followed by a restart of the last bootstrap loader. This restart will load new code and/or a new configuration file if these were changed.
`COLDBOOT`	Orderly shutdown, followed by a power on hardware reset (which eventually causes a boot of the system.

Each of these commands performs the following functions:

1. Complete all outstanding I/O operations and quiesce the I/O subsystem.

2. Switch to the boot processor.

3. Stop and reset (initialize) all other processors.

4. Close the system log and save state information to the internal hard drive (if any).

5. Save any application *pinned data* --data that applications were unable to write to physical devices due to a hardware error (such as a power failure).

6. Restart the system, or halt processing (stop). In stopped state, a subset of RTOS commands can still be entered – including restart, coldboot, and warmboot.

Quiesce and Resume Commands

RTOS supports the ability to *quiesce* the system – causing the system to finish all outstanding I/O operations then stop, waiting for a *resume* command to continue operations. These commands are used when an operator wishes to change hardware attached to the system, such as adding or deleting devices.

Power Failure Handling

RTOS can monitor one or more attached *Uninterruptable Power Systems* (UPS). The UPS may be attached to a serial port, or to a platform specific UPS interface.

If the UPS reports an alternating current (main power, or *AC*) failure, the system enters *synchronous mode,* where applications do not receive successful data write completion status until data are physically written to physical devices.

If the system receives *battery low* from the UPS, the system performs an implicit shutdown (see above). The system can also be configured to shutdown if a certain time has elapsed without restored AC power.

Since the UPS may make power available to an RTOS platform longer than to attached peripherals, the system supports the concept of *pinned data*. These are data that an application has written in *asynchronous mode* – a mode where the application is given successful write completion status when the data are cached, rather when the data are written to a device. These data cannot be physically written to a device because of a hardware error (in this case, loss of power), thus they are pinned in memory.

These data are saved to an internal hard drive (if installed) and restored and written when power returns and the system is restarted.

System Basics and Terminology

In the following text, I've tried very hard to neither define new terms, nor redefine existing ones. That said, just like natural language, computing terms often take on different meanings over time, so I'll quickly describe some of the terms I'll be using in this paper.

The terms *data structure*, *structure*, and *control block* are used interchangeably to mean a contiguous storage area containing related data. The terms *objec*t and *field* refer to a single data element or data array within a data structure, such as a byte, pointer, or smaller imbedded data structure. *Bytes* are individually addressable eight bit entities (sometimes called *octets* -- but not by **this** author). Multi-byte objects are referred to by specific size (for example, a four-byte object), rather than by the platform dependent terms *word*, *halfword*, *fullword*, and so on (as these terms tend to be system dependent).

Routine, *subroutine*, and *function* are used somewhat interchangeably to mean called code that usually performs an atomic operation, then returns to the instruction following the call.

In C code prototype examples, native C data types (rather than defined types) are used.

The term *NULL* refers to a pointer to nothing. RTOS always checks the validity of a pointer by comparing a pointer to NULL (rather than to zero).

The term *task* refers to a single execution thread sharing resources with other system tasks. The term *application* refers to a set of one or more tasks that together perform a given function. For example, a task might handle incoming IP messages, while a TCP/IP application might handle an active FTP user.

General Data Structure Layout

A common format is defined for all system control blocks. Each control block begins with a four-byte *eye-catcher* (further described in *Debugging* below), optionally followed by linked list queuing pointers (also described below). Objects within a structure are grouped by size to optimize storage utilization and take advantage of any performance improvements resulting from optimal object boundary alignment. As much as possible, objects are grouped by similar function to make it easier for developers and debuggers to find them.

In virtually all cases, control blocks are of fixed length and, if dynamically allocated, are placed in a storage block that is a power of two in byte size. A portion of RTOS initialization code verifies that each control block completely fits within the storage space allocated for it. All common control blocks have kernel and possibly task routines defined to allocate and initialize, or release them.

Basic Control Block Initialization (BASE)

The RTOS maintains a top-level data structure called the base control block, or *BASE*. This block is used as the base of many data structure linked lists, contains overall system flags, locks, constants, and counters. Virtually all control blocks can be located starting from the base block, consequently, most system function calls pass a pointer to this BASE block.

Certain BASE block fields are initialized by the bootstrap loaders. These locations are kept in a fixed location at the start of the block so that the RTOS and the bootstrap loader can inter-operate even at different version levels.

Generally, only the kernel and privileged tasks can modify the base block. All tasks can **read** data from the base block.

Queuing Functions

Control block queuing was deemed to be **absolutely fundamental** to the basic design of the operating system. Queuing functions were developed before anything else in the RTOS.

Queuing routines give the kernel and tasks the ability to atomically, in a multiprocessing-safe manner, add, delete, reorder, insert, and remove blocks from singly and doubly linked lists.

Presently, only the doubly linked list routines are used. The small amount of additional storage and processing overhead caused by double linking justifies the flexibility and reliability provided by the doubly linked list routines.

Two data structures support linked lists, the base of the list (called a *queue header* or *QHEAD*), and a list entry (called a *queue element* or *QELEM*). Each queue **header** structure is generally imbedded into a higher level structure, for example, a device block would contain the base of a linked list of active I/O operations.. Each queue **element** structure is imbedded in the block to be queued, for example, an I/O operation block for an I/O operation is an element of a list of active I/O operations.

A queue header has the following contents:

- A pointer to the first element on the list, NULL if an empty list.
- A pointer to the last element on the list, NULL If an empty list.
- A count of elements presently on the list, zero if an empty list.
- A multiprocessor lock, zero if unlocked.
- A pointer to this queue header.

A queue element has the following contents:

- A pointer to the next element, NULL if none.
- A pointer to the previous element, NULL if none.
- A pointer to the queue header, NULL if not queued.

The queue header pointer field is used to verify that the header is indeed a queue header. In queue elements, the queue header pointer is used to indicate whether the block is already (or not) queued, and, if queued, that it is queued on the correct list.

Queue header pointers are **only** used by the queuing functions for validation, **and are never used by an application or the system to identify a block's queue.** In other words, as a double consistency check, a routine that manipulates linked lists should independently be aware of a queue element's status.

These routines are (thankfully) robust to the point that, in the entire development of this system, no problem was ever caused by undetected queue inconsistencies, thus completely avoiding a very common class of failure (especially in multiprocessing systems).

The following list routines used to manipulate queues. Generally, the arguments for a routine passing a data structure are a pointer to the data structure, a pointer to the queue header, and the offset within the data structure to the queue element data:

```
void qinit (struct QHEAD *);   // Initialize queue base block
void qhs (void *, struct QHEAD *, long int);
                  // Queue to head of singly linked list
void qts (void *, struct QHEAD *, long int);
                  // Queue to tail of singly linked list
void * dqs (struct QHEAD *, long int);
                  // Dequeue from head of singly linked list
long int qhd (void *, struct QHEAD *, long int);
                  // Queue to head of doubly linked list
                  // Return NUMBER of blocks queued INCLUDING this one
long int qtd (void *, struct QHEAD *, long int);
                  // Queue to tail of doubly linked list
                  // Return NUMBER of blocks queued INCLUDING this one
long int qtdlh (void *, struct QHEAD *, long int);
                  // Queue to tail of doubly linked list
                  // It is assumed that the queue lock IS ALREADY HELD
long int qifd (void *, void *, struct QHEAD *, long int);
                  // Insert after block in doubly linked list
                  // Return NUMBER of blocks queued INCLUDING this one
long int qibd (void *, void *, struct QHEAD *, long int);
                  // Insert before block in doubly linked list
                  // Return NUMBER of blocks queued INCLUDING this one
void * dqhd (struct QHEAD *, long int);
                  // Dequeue from head of doubly linked list
void * dqtd (struct QHEAD *, long int);
                  // Dequeue from tail of doubly linked list
void * dqd (void *, struct QHEAD *, long int);
                  // Dequeue from doubly linked list
void * qmhd (void *, struct QHEAD *, long int);
                  // Move an already queued block to the head of a
                  // doubly linked list
                  // The entire operation is performed holding the
                  // queue lock
void * qmtd (void *, struct QHEAD *, long int);
                  // Move an already queued block to the tail of a
                  // doubly linked list
                  // The entire operation is performed holding the
                  // queue lock
void * qmtdlh (void *, struct QHEAD *, long int);
                  // Move an already queued block to the tail of a
                  // doubly linked list
                  // It is assumed that the queue lock IS ALREADY HELD
void * dqdlh (void *, struct QHEAD *, long int);
                  // Dequeue from doubly linked list
                  // It is assumed that the queue lock IS ALREADY HELD
```

System Support Functions

Display routines

Display routines display messages of varying importance (normal, logged, time-stamped, input-echo, and so on). These routines automatically direct messages to the intended display surface, whether it be a local console, a Telnet session, a serial port (dial-up), or a remote system. These routines are always available to any caller, whether in kernel mode, in a task, or even if the system has crashed and is being debugged..

The RTOS display screen format is very basic by today's graphical user interface standards, and reminds one of old mainframe displays – no graphics at all! There is a title line at the top, a number of upwardly scrollable output lines, and a user input line or two at the very bottom. (we plan to provide GUI support via HTTP/HTML services described below).

When a display routine is called, the message contents are queued for subsequent display, log, etc. Permanently resident tasks route and display queued messages.

Message display routines pass a pointer to a zero-byte-terminated character string.

Some display routines include:

```
void msgclear  (struct BASE *);           // Clear the output area
void msgclrinp (struct BASE *);           // Clear the input area
void msgscroll (struct BASE *);           // Scroll output area up 1
void msgwrite  (struct BASE *, char *);  // Write one line to output
void msgwriteb (struct BASE *, char *);  // Write one bright line
void msgwritee (struct BASE *, char *);  // Write echoed input line
void msgwriteg (struct BASE *, char *);  // Write remote message line
void msgwritet (struct BASE *, char *);  // Write title line
void msgwrdati (struct BASE *, char *);  // Write date/time to title
void msgwrstat (struct BASE *, char *);  // Write status info to title
void msgwritei (struct BASE *, char *);  // Write to cmd. input area
```

Numeric Conversion Routines

RTOS routines convert binary values of varying sizes to displayable decimal and hex.

The absence of *printf*-like routines has caused some developers to grumble, while they go back to counting on their fingers to calculate offsets to get numbers correctly plugged into messages. We felt that scanning and formatting should be done once (by the developer) and not take up runtime cycles every time a message is displayed.

Routines that convert from binary to character format pass the value and a pointer to the output buffer (which is assumed to be large enough to contain values of the passed type). Routines converting character to binary pass a pointer to the character string, the length in bytes of the character string, and a pointer to the location to receive the converted binary result. The returned value indicates whether the conversion was successful. Here are some representative numeric conversion routines:

```
void cvtbdb (char, char *);          // Convert byte to decimal
void cvtbdw (short int, char *);     // Convert two bytes to decimal
void cvtbdd (long int, char *);      // Convert four bytes to decimal
void cvtbhb (char, char *);          // Convert byte to hex
void cvtbhw (short int, char *);     // Convert two bytes to hex
void cvtbhd (long int, char *);      // Convert four bytes to hex
void cvtbhdl (long int, char *);     // Convert “other Endian”
                                     // four byte value to hex

short int cvtdbb (char *, short int, char *);   // Cvt decimal to byte
short int cvtdbw (char *, short int, short int *);  // Cvt decimal
                                     // to two byte binary
short int cvtdbd (char *, short int, long int *);  // Cvt decimal
                                     // to four byte binary
short int cvthbb (char *, short int, char *);   // Convert hex to byte
short int cvthbw (char *, short int, short int *);
                                     // Cvt. hex to two byte binary
short int cvthbd (char *, short int, long int *);
                                     // Cvt. hex to four byte binary
```

Command and File Parsing Routines

These routines:

1. Parse command, configuration files, and other textual input.

2. Do case insensitive string compares.

3. Call a command handler given a command verb (supporting abbreviations and shortened forms).

4. Require a user to enter a line of input (such as a confirmation or a masked password).

5. Check a user's authorization to issue a command.

These routines operate on a structure called a *command block* (*CMDB*). This structure holds a string, and pointers and lengths relating to that string. Input device drivers (such as the keyboard driver) allocate and fill in a CMDB each time a user enters an input string.

These routines are used in command handling:

```
void parmscaninit (struct CMDB *);
                          // Initialize string for parameter scanning
unsigned char * parmscan (struct CMDB *);
                          // Return next non-blank string
unsigned char * linescan (struct CMDB *);
                          // Return next CR/LF delimited line
void parmsave (struct CMDB * newcmdb, struct CMDB * oldcmdb);
                          // Save parse state
void parmrestore (struct CMDB * oldcmdb, struct CMDB * newcmdb);
                          // Restore parse state
short int parmonoff (struct CMDB *);
                          // Get and return whether next
                          // parameter is the word "on" or "off"
short int upparmget (struct CMDB *);
                          // Get next parameter "upper-cased"
char casecmp (unsigned char *, unsigned char *, long);
                          // Case insensitive compare of two strings
                          // of the length in parameter three.
                          // RC = 0 if equal, one otherwise
struct CMDT * findcmd (struct CMDB *, struct CMDT *);
                          // Return command address given CMDB cmd
char verifypriv (struct BASE *, struct TB *, char);
                          // Verify user's authorization to run something
                          // based on passed privilege field
char verifycmdpriv (struct BASE *, struct TB *, struct CMDT *);
                          // Verify user's authorization to issue
                          // this command/subcommand
struct CMDB * keybgetinput (struct BASE *, unsigned long int);
                          // Return a line of input to a task
short cmdtaskqueue (struct BASE *, struct CMDB *);
                          // Routine to run a command
                          // from a task as a task
short cmdtaskrun (struct BASE *, struct CMDB *);
                          // Routine to run a command
                          // from a task as a subroutine
```

Other Useful Routines

Other useful routines include:

```
char bitstring (unsigned long int, char *, char *);
                       // Given four byte bitstring, return leftmost
                       // bit number (as bit zero) and the number of
                       // contiguous bits in the string
long int andstring (unsigned char *, unsigned char *, long int);
                       // Check string against AND mask
short int bitscanw (short int);  // Find leftmost bit in two bytes
short int bitscand (long int);   // Find leftmost bit in four bytes
short int bitscanrw (short int); // Find rightmost bit in two bytes
short int bitscanrd (long int);  // Find rightmost bit in four bytes
unsigned long int byteswap (unsigned long int);
                       // LE four bytes into BE order (and vice versa)
unsigned short int byteswapw (unsigned long int);
                       // LE two bytes into BE order (and vice versa)
```

Storage Management and Allocation

Virtual Memory

All memory is addressed such that its virtual address is equal to its real address (*V=R*). The system runs with translate enabled to take advantage of protection and caching control that would otherwise not be available.

Since the largest contiguously allocable data buffer is 4,096 bytes (the size of a page), the I/O subsystem has the ability to generate *scatter-gather* (*SG*) lists to do I/O to larger logically contiguous regions.

The current RTOS applications do not require more addressable virtual memory than physical memory (at the time this paper is being written, physical memory is **extraordinarily** inexpensive). The initial hardware platform has no practical limit on the amount of storage than can be physically installed), so there is presently no code to support multiple address spaces or paging. Adding this would not be a significant effort – this would most likely be done if we port RTOS to an architecture with the provision for much more virtual than affordable real memory.

Storage Protection

All unused or uninstalled memory is marked *invalid* in the system translate tables, resulting in a hardware exception if it is inadvertently addressed.

Sensitive control blocks, such as the base block are accessible to all tasks, but can only be modified by the kernel or appropriately privileged tasks.

Individual 4,096 byte pages of memory can be marked as *uncacheable* (certain I/O devices require this if they don't (correctly) support cache coherence protocols).

Private memory (memory belonging to a specific process) is also marked invalid to both tasks and the kernel. In a storage controller application, this includes all user data (as the system and application moves this data to and from I/O devices, but has no reason to access its contents).

Pages that are broken down into smaller, equal sized pieces for control blocks, **are** accessible to applications. To date, as system functions are used to initialize these data structures, we have experienced no errors resulting from one task overwriting another tasks' control blocks. While it is obviously possible and sometime useful to allocate pages by task, our applications tend to pass blocks from task to task, requiring that tasks share this memory.

Storage Allocation and Release

Dynamic allocation and release of memory for various kernel and task structures are performed by a set of routines that work with fixed-sized blocks of storage. At system initialization, an initial pool of each storage size is created. Requests for storage are filled from these pools. If a pool is empty, a 4,096-byte page of memory is allocated and broken into an integral number of pool-sized blocks.

When a block is released, it is returned to its pool. If the pool grows beyond a configured limit, contiguous free elements are reclaimed from the pool and recombined back into a free 4,096-byte page.

If storage is not available to a kernel routine requester, the requesting routine receives a NULL pointer. Thus kernel routines are designed to recover from insufficient storage conditions.

Tasks have the option of implicitly waiting until storage becomes available, or they can receive a NULL pointer when storage is not available to fulfill the request.

Because free storage pools are maintained as doubly linked lists of storage elements, storage allocation and release are very fast. Queue consistency checking allows for quick discovery of routines misusing or releasing already released storage blocks.

Current storage pool sizes are (in bytes): 64, 128, 256, 512, 1K, 2K, and 4K. Blocks of each size are aligned on their respective boundaries (providing another consistency check).

There are two special-purpose storage allocation routines, normally only used at system initialization or by initializing device drivers. The first, *gethuge*, will allocate a contiguous real memory buffer of *n* 4,096-byte pages on a 4,096-byte boundary. The second, *getaligned4k*, will allocate storage on a specific boundary. These routines fulfill the needs of device drivers that require large or specially aligned buffers.

Here are some examples of storage allocation routines:

```
//  Routines for task callers
char * get64b (void);                   // Allocate 64 byte block
char * get128b (void);                  // Allocate 128 byte block
char * get256b (void);                  // Allocate 256 byte block
char * get512b (void);                  // Allocate 512 byte block
char * get1kb (void);                   // Allocate 1K block
char * get2kb (void);                   // Allocate 2K block
char * get4kb (void);                   // Allocate page
void fret64b (char *);                  // Release 64 byte block
void fret128b (char *);                 // Release 128 byte block
void fret256b (char *);                 // Release 256 byte block
void fret512b (char *);                 // Release 512 byte block
void fret1kb (char *);                  // Release 1K byte block
void fret2kb (char *);                  // Release 2K byte block
void fret4kb (char *);                  // Release page
char * get2kbc (void);                  // Allocate 2K block (maybe)
char * get4kbc (void);                  // Allocate page (maybe)

//  Routines for system callers
char * gethuge (struct BASE *, long int);   // Allocate HUGE contiguous,
                                        // cache block aligned storage
                                        // (rounded up to a cache block
                                        // size)
char * getaligned4k (struct BASE *, unsigned long int);
                                        // Get specifically aligned
                                        // block of storage
char * get64 (struct BASE *);           // Allocate 64 byte block
char * get128 (struct BASE *);          // Allocate 128 byte block
char * get256 (struct BASE *);          // Allocate 256 byte block
char * get512 (struct BASE *);          // Allocate 512 byte block
char * get1k (struct BASE *);           // Allocate 1024 byte block
char * get2k (struct BASE *);           // Allocate 2048 byte block
char * get4k (struct BASE *);           // Allocate 4K page
void rel64 (char *, struct BASE *);     // Release 64 byte block
void rel128 (char *, struct BASE *);    // Release 128 byte block
void rel256 (char *, struct BASE *);    // Release 256 byte block
void rel512 (char *, struct BASE *);    // Release 512 byte block
void rel1k (char *, struct BASE *);     // Release 1024 byte block
void rel2k (char *, struct BASE *);     // Release 2048 byte block
void rel4k (char *, struct BASE *);     // Release page
```

The kernel and/or any task can call any of these routines.

Almost all storage routines are called by higher-level routines used to allocate and release **specific** control blocks, as described below. This convention isolates the task from the size and initialization of a given structure, and tends to minimize errors caused by accidentally over-subscripting one's data blocks.

Control Block (Structure) Allocation and Release

To simplify and centralize the allocation and release of commonly used structures, system routines are provided to allocate and initialize, or release each major control block.

Allocation routines allocate storage, initialize the structure's fields, and possibly add the block to one or more appropriate linked lists.

Structure release routines verify the passed control block's identity, possibly release linked lists of subordinate control blocks, possibly remove the block from one or more linked lists, and release the storage.

The benefit of such routines are that they remove the need for duplicate, probably inconsistent application code. They also remove the need for an application to be aware of the size of a data structure. An obvious alternative to these routines would have been to create C++ classes with constructors and destructors – however, concerns with performance and portability caused us to choose to not use C++ features (other than the C++ comment format).

Here's an example showing the routines that allocate and release I/O control blocks:

```
struct IOB * getiob (struct BASE *, struct RDEV *);
                            // Allocate an I/O block
void filliniob (struct IOB *, struct BASE *, struct RDEV *, struct TB *);
                            // Fill in a passed I/O block
void reliob (struct IOB *, struct BASE *);
                           // Release an I/O block
```

Generally, there are no alignment requirements for a control block, so some tasks use a portion of their stack space as a control block. The above routine *filliniob* could be used to initialize an "in stack" control block, or to re-initialize an existing control block.

Multitasking and Task Services

This section describes some of the most common task and application support provided by the kernel.

The RTOS has three execution privilege levels – *kernel mode* (sometimes known as *supervisor state*), *privileged task mode*, and *normal task mode* (together sometimes called *user state*). All device driver code and the basic operating system run in kernel mode. Privileged tasks can access system control blocks and can directly issue I/O instructions (manipulate I/O memory and I/O registers).

All task code, and generally all kernel code runs enabled for interrupts. Certain very small segments of code are completely disabled for interrupts, and, in the case of a device driver interrupt handler, may mask pending lower priority interrupts.

Each RTOS task is controlled and referenced by a task control block (*TB*).

Task Support

The kernel provides a myriad of task support routines or *system services*. A developer invokes these as subroutine calls, however, in most cases they result in kernel service requests by issuing a *software interrupt* (also known as *supervisor call*, *system call*, *INT*, and so on). Since the specific nature of system software interrupt calls are hidden from tasks and applications, they can be renumbered or have their parameters changed with no effect on application code. This is beneficial when porting code, as different hardware platforms may have a different number of potential software interrupt slots.

Some support routines are restricted only for use by privileged tasks. There are no unprivileged-task calls to privileged tasks, and there are no kernel to kernel interrupts (well, **almost** none – see Multiprocessing below).

Tasks **always** run with the processor enabled for interrupts.

Task Wait and Post

Any task can wait for one or more events to take place via the use of a *WFB* (*wait-for block*). Either the kernel, or another task can post completion of one or more of these events and return an individual one byte *return code* for that event. The wait/post logic *or's* the return codes, so that the calling task can quickly determine whether **any** posting entities set a non-zero code (indicating an error).

Alternately, a task can wait for a counter value to be decremented to zero. Each entity posting a *counter WFB* decrements the counter – when it goes to zero the waiting task is dispatched.

Because of task scheduling and multiprocessing, it is possible for a WFB to be posted **before** the task actually goes into a wait state, in which case the task simply continues to run.

Here are some samples of wait/post routines (note that some of these are described in the following sections:

```
void * wait (unsigned short int); // Wait till event(s)complete
void * waitcount (unsigned short int);
                               // Wait till counter goes to zero
void * waitpb (struct WFB *);  // Wait till event(s)
                               // complete - call with passed WFB
void post (struct TB *, unsigned short int, unsigned short int);
                               // Flag event complete
void postwfb (struct WFB *, unsigned short int, unsigned short int);
                               // Flag specific WFB event complete
void delay (unsigned long int);   // Suspend for n TIMEUNITS secs.
short int waitpbtimeout (struct WFB *, struct TDB *, long int);
                               // Task wait on event AND/OR
                               // time expiration (timeout)
```

Timed Events and Delays

Timed events are a specialized case of wait/post logic. A task can delay execution (wait) for a specified period of time. The timer resolution for delays (in this version of RTOS) is 512ths of a second.

Generally, tasks pause to either let an amount of time pass before resuming work, for example, to delay before trying to access a failed resource such as a remote connection, or to do a certain amount of work at regular intervals, for example, to update system statistics every "n" seconds.

Wait with Timeout

It is possible to combine wait/post with a time delay value. This gives you the ability to wait for one or more events, but to continue execution if a time interval expires before completion occurs.

For example, many systems (including this one) drop Telnet connections if there has been no user interaction for a period of time. Also, TCP protocol requires retransmission of packets if an acknowledgement has not been received. Both these cases are handled via "wait with timeout" logic.

All system timing events are handled via a time delay block (TDB). Kernel routines can also start timed events and receive control when they expire (for example, this is how missing interrupts are detected and handed).

Server Tasks

Tasks that are created to process work on behalf of the kernel, device drivers, or other tasks and applications are called *server tasks*..

For example, a received TCP/IP packet may be queued by a device driver to be handled by a task (**which** task might depend on the origin IP address and the message type). Non-critical or interruptible work can also be handled by tasks on behalf of the kernel or a device driver, for example, to log console messages to a disk log file. Tasks may also pass work on to other tasks (for example, one task may route work to other tasks based on their content).

Server tasks can be used to serialize work and thus guarantee an ordering of events. For example, a data destaging task might be used to insure that data are written to a physical device in the order sent by the application. Such a task would also have the ability to scan pending work and combine requests to optimize operations, for example, combining a number of pending sequential writes into one larger one.

Server tasks work by waiting for control block(s) to be queued to their task block. The value passed by the server task defines the offset within the queued control block(s) where the linked list queue pointers reside. The server task implicitly dequeues the top entry, processes it, releases it, and loops back to handle the next queued entry (if any). When the list is empty, the server task waits for another entry to be queued.

Like wait/post, it is possible for a server task to time-out waiting for more work to handle – this can be used to take action when an anticipated event did not occur in the expected time period..

Here are routines called **by** server tasks, and for others to queue work **for** server tasks:

```
void * waitforwork (long int); // Called by a server task to wait for
                               // something to process
void * waitforworkortimeout (long int, struct TDB *, long int);
                               // Called by a server task to wait for
                               // something to process or for
                               // a time to expire
void queuework (struct TB *, void *);
                               // Called to place a block on a server
                               // task's queue
```

Spawning Child Tasks and Waiting for Child Tasks to Complete

Each task is called as a *C* routine with up to six passed parameters. A task is entered with the same calling sequence as any other *C* routine – its implicit caller is set to the system's end-of-task handler.

The kernel or any task can *spawn* (create) additional tasks. A task spawned by another task is called the *child* to that *parent* task.

A task can either wait for a spawned task to be created, or can be told that there are insufficient resources to spawn a task. Since the calling sequence for a task and a subroutine are identical, a parent task might simply call the child task code as a subroutine if there are insufficient resources to start a child task.

A task can wait for all spawned tasks to complete. For example, in a mirrored data application, tasks are spawned to simultaneously issue the individual device writes – the parent task waits for these child subtasks to complete, then checks their return codes to insure that the writes were successful.

These routines are used to spawn tasks, and wait for child tasks to complete. Note that there is no need for a routine to post child task completion – this is implicitly done by the kernel.

```
struct TB * spawntask (short int, void *, void *, long int [6]);
                               // Create and run a task
char spawntasknw (short int, void *, void *, long int [6]);
                               // Conditionally create and run a task,
                               // return if we can't
void waittasks (void);         // Wait for all created tasks to complete
```

Other System Services

Here is a sample of other noteworthy system services, such as date and time routines, pseudo random number generation and debugging trap calls:

```
void datetime (char *);      // Get current local date/time string
void httpdate (char *);      // Return HTTP format GMT printable date
void gmtdate (char *);       // Return GMT printable date
void addtime (long int *, long int);  // Calculate expiration time
void timestamp (long int *); // Timestamp a location

void randinit (unsigned long int, unsigned short int [250], unsigned
short int *);      // Initialize random number generator
void randinit32 (unsigned long int, unsigned long [250], unsigned short
*);                // Initialize random number generator
unsigned short int rand (unsigned short int [250], unsigned short int *);
                   // Return random number from 0 to 65535
unsigned short int randn (unsigned short int, unsigned short int [250],
unsigned short int *); // Return random number from 0 to n
unsigned long int randd (unsigned long [250], unsigned short int *);
                   // Return long random number from 0 to (2**32)-1
unsigned long int randnd(unsigned long int, unsigned long [250], unsigned
short int *);      // Return long random number from 0 to n

void fatalerror (long int);  // Routine to croak system w. ecode
void stopsystem (long int);  // Routine to stop the system w. ecode
```

Multiprocessing (MP) and Dispatching

RTOS supports symmetric and asymmetric multiprocessing. In *symmetric multiprocessing*, all running processors are capable of running tasks and handling I/O interrupts. In *asymmetric multiprocessing*, only one (or a set of) processors handle I/O interrupts. Normally a system is configured to start all working processors in symmetric mode.

Locks

To prevent simultaneous access to serialized system resources, RTOS provides multiprocessing-safe *locks*. While one processor holds a lock, other processors cannot obtain it. Thus, lock and unlock calls are placed around MP-sensitive code sections. A processor that tries to obtain an existing lock *spins* (loops) until that lock is released or until the dead-man timer causes an interrupt..

Obtaining a lock disables the local processor (and returns the previous processor enable state). The subsequent unlock call restores the previous processor enable state. This disable/enable sequence prevents interrupt handlers on this processor from attempting to obtain a lock held by the **locking** processor.

Here are some of the considerations for using locks:

1. Locks should be held for the shortest time possible.

2. Lock/unlock should surround the **minimum** code necessary to perform the serialized function. Or, expressed another way, locks should be obtained for the smallest possible resource, for example, a device register versus an entire device, a device versus an entire bus.

3. As all tasks run enabled, tasks cannot unconditionally obtain locks (but they can conditionally obtain locks – see below).

4. In only very rare circumstances should more than one lock be obtained at a time, and in no case should multiple locks be obtained in a way that could result in a deadly embrace condition between processors (where one processor holds a lock that the other needs and vice versa).

In addition to unconditionally obtaining and releasing a lock, there is a call to conditionally obtain a lock, and return whether the lock was obtained. This routine **can** be used by tasks, as it does not disable the local processor.

There are also routines to atomically update storage values. These are used to obtain unique values, such as task identifiers, statistical counters, and so on.

This implementation reserves one byte for each lock, although the actual locking flag is a single bit.

Here are representative lock functions:

```
char testandset (unsigned char *);
                       // Task conditionally obtain a lock
unsigned long int lockit (void *);
                       // Lock a control block for exclusive use
char clockit (void *); // Conditionally lock a control block for
                       // exclusive use
                       // Returns 1 if locked, zero if lock already held
                       // Does NOT change (or return) enable state
unsigned long int locknw (void *);
                       // Lock a control block without waiting
void unlockit (void *, unsigned long int);
                       // Unlock a locked control block
void cunlockit (void *);  // Unlock a locked control block without
                       // changing enable state
unsigned long int lockedupdate (unsigned long int *, long int, char *);
                       // Interlocked doubleword update
                       // Returns ORIGINAL value
unsigned long int lockedincrement (unsigned long int *, long int, char
*);                    // Interlocked doubleword increment
                       // Returns ORIGINAL value
```

Task dispatch

At creation, each task is given one of four priorities, *high*, *medium*, *low*, and *none* (you might correctly conclude from the silly fourth priority name, there were originally **three** priorities). If a task is eligible to run, that is, it is neither waiting for resources nor an event to complete, and isn't already running on another processor, it will be dispatched if no higher priority task is eligible to run.

A runnable higher priority task will always preempt a lower priority task. Within a priority, tasks are added to the bottom of the dispatch list. The task on the top of the highest priority dispatch list containing eligible tasks will be the task next dispatched.

Each task is assigned a time-slice value. Each time the task is dispatched, this value is loaded into a timer register which auto-decrements and interrupts the processor when it reaches zero. If a task is preempted or voluntarily waits, its remaining time-slice is saved and used at the next dispatch. If a task's time-slice reaches zero, the task is moved to the back of its dispatch priority list and is given a new, full time-slice.

This means that lower priority tasks are always preempted by higher priority tasks, but within a priority, tasks receive a *fair share* of processor resource before being preempted. Besides balancing the task load, this prevents any task from hogging the entire machine.

Resource Wait

If a task requires resources that are not available (most notably storage), the task is placed into *resource wait*. We refer to this task as being *suspended*. The task is queued on the back of a list of tasks waiting for that resource (or resource class). When the resource becomes available (in the case of storage, a block of the correct size becomes available), the resource is given to the task, and the task is returned to the dispatch list.

Resources are granted to tasks in task request order, regardless of task priority.

Besides voluntary wait conditions, a task can be suspended for any of the following reasons:

1. No storage of a given size.

2. Insufficient resource to create a subtask (storage, or maximum simultaneous task value exceeded).

3. Application-created resource lock condition (for example, in a storage controller product, waiting for shared or exclusive use of a cached data page).

Affinity

By default, a task will run on any available processor. It is possible to assign a task to a specific processor, or switch which processor is handling a task, for example, to serialize an activity such as a flash memory update, to access processor specific registers, or to load balance activity.

Here are routines to set task affinity:

```
void setbootfocus (void); // Run task on BOOT processor
void settaskfocus (unsigned short int);
                          // Run (and switch) task on specific processor
void cleartaskfocus (void);  // Run task on ANY processor
```

Inter-Processor Signaling

Each processor can dispatch tasks and handle interrupts. If a processor finds no work to do, it goes into a *wait state* – a stopped state where the processor is enabled for interrupts but is processing no instructions. The wait state ends if either of the following events occur:

1. The processor is interrupted by a hardware interrupt (usually an I/O interrupt).

2. Another processor *wakes* this processor via an inter-processor interrupt. Generally, the only time one processor wakes another is if a dispatching processor detects that there are multiple tasks waiting to be dispatched.

When a waiting processor wakes, it checks the dispatcher and resource waiting lists. If any are non-zero, it parcels out resources to waiting tasks (if any), then runs the first dispatchable task.

The boot processor processes timer and local keyboard interrupts. Any on line processor can process all other interrupts. As a result, the boot processor is generally the one that detects excess work and wakes up other processors.

Timer Processing

The interval timer (a timer that interrupts the processor at a regular interval) is assigned the highest I/O interrupt priority on the system. The timer interrupt handler increments an eight byte value in low storage. This value is compared against the queue of time delay blocks (*TDB*), which are sorted into ascending order. Any TDB's that have a lower or equal time value are processed and either removed from the queue or rescheduled.

The timer interrupt handler also reloads the dead-man timer (if any), checks the power failure duration, and optionally logs the interrupted instruction address (for RTOS and application profiling).

The current implementation increments the in-storage timer value once every 512th of a second. RTOS and ASC commands referencing timer interval values are expressed in units of 512ths of a second.

Time of Day Processing

The RTOS uses clock hardware to obtain the current date and time. The clock is set to UTC (or GMT) – the RTOS handles time zone offsets. The RTOS also allows the operator to update the time and/or date.

The clock hardware does not generate any interrupts. Every five seconds, a task reads the clock information and displays it on the console. Any time-stamped messages also read the value from the clock hardware.

I/O Subsystem

This section describes the I/O subsystem. The I/O subsystem is responsible for queuing I/O, starting I/O, and posting I/O operation completion to waiting tasks.

Normally, tasks request I/O operations. The kernel has the ability to start I/O, but rarely does so, for example, the kernel can snapshot memory to disk for subsequent debugging if a fatal system error occurs.

The Real Device Model

The I/O subsystem manages *devices* and *device controllers*. A device controller (for example, a SCSI bus controller) may control multiple devices.

At initialization, a device driver sets the maximum number of *I/O units* it can simultaneously handle as a controller, and, separately, the number it can handle for each individual device. For example, a controller might have a fixed number of internal entries that limit the number of simultaneous I/O operations that can be outstanding. A device might also have a limit of the number of simultaneous operations it can perform. For SCSI devices, this would be the maximum number of tagged queuing commands it can support.

When a task requests that an I/O operation start, the device driver is called to determine how many controller and device I/O units this I/O operation will need. If there are sufficient available device and controller units, the I/O is started, and the number of active device and controller I/O units is updated. Otherwise the I/O operation is queued on either the device or the controller (depending on which has run out of resources).

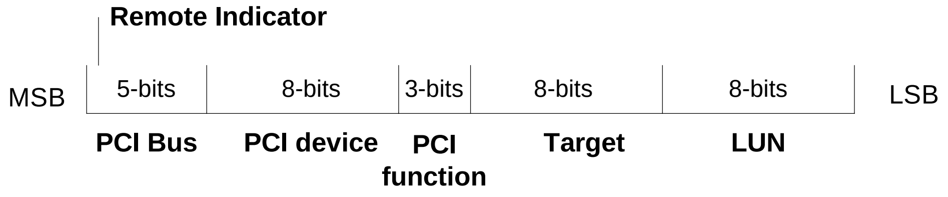
As the completion of each I/O operation, the number of active I/O units is again updated. If device resources are now available, I/O operation(s) waiting on the device are propagated to the controller, and if controller resources are available, I/O operation(s) waiting on the controller are started.

The RTOS I/O subsystem (rather than the various device drivers) handles resource allocation.

The I/O subsystem supports multiple physical paths for any physical or emulated device. This potentially improves performance and improves reliability (if an individual path fails, remaining paths are used).

Device Addressing

Local devices and controllers are assigned unique four-byte numeric addresses based on their hardware (usually PCI) bus number (five bits), bus device number (eight bits), and function (three bits), and their target (eight bits) and logical unit number (eight bits). *Remote devices* are additionally referenced via the IP address (or name) of the system they are physically attached to, and their numeric address on that system.



Externally (say, by an operator command), devices can be referred to by either a hex number (as described above) or by a configuration-assigned symbolic device name. Further, devices can be referred to by their unique volume serial number (if one exists).

For example, device 380104 refers to target 1, logical unit 4, on the controller plugged into PCI bus 0, PCI device 7, PCI function 1. If the configuration file equated the name *George* to the local device *380104*, then commands could use either the number or the name, and displays would, in most cases, display the name *George*.

Device names help to eliminate operator errors resulting from typos in input commands.

I/O operations

The I/O block (*IOB*) is passed by a task to request an I/O operation. It contains many things, including a pointer to the Real Device Block (*RDEV*) for the appropriate device, information about the device command (for SCSI and IDE devices, this is a SCSI Command Descriptor Block (*CDB*)), and either a buffer pointer and buffer length, or a pointer to a list of buffers and lengths, and a count of these buffer/length pairs (this list is often called an *SG list*).

A task can either wait for an I/O operation to complete, or specify that the task continues while the I/O is active. A task can also signal that it wants to be notified of an error (without certain error recovery implicitly taking place). For example, tasks that scan busses for devices need to turn off I/O timeout recovery as most bus addresses will not have devices installed.

Each IOB has an expected *I/O duration value*, specifying the amount of time the I/O is expected to take (with a reasonable margin added). Default values are set automatically based on device type. These can be overridden by the requestor (for example, a disk format takes much more time than a disk block read). If the operation does **not** finish within this time (with margin figured in based on the other I/O activity in process), a *missing interrupt* event will take place (see below).

At the conclusion of an I/O operation, the task receives a *return code*, and a residual count (number of bytes **not** transferred). In the case of device-reported errors, the *sense data* (device dependent data describing the nature of the error condition) are automatically collected and returned as part of the I/O operation. A separate sense operation is not required.

There are a number of special I/O requests, mostly related to error recovery. A task can request a *bus reset*, a *controller reset*, a *device reset*, or a *special controller operation* (usually used to collect device dependent statistical information, or to perform some controller-dependent function such as fibre channel fabric name search).

The I/O subsystem verifies that data buffer addresses and lengths are valid. The I/O subsystem also collects controller and device statistics on the number of I/O operations performed, and the number of bytes transferred.

Missing and Hot Interrupt Handling

The I/O subsystem handles missing interrupts by starting a timer based on the expected ending time of the I/O operations active on that controller. If this timer expires before the active I/O finishes, the device driver is called by the missing interrupt handler (*MIH*) with a pointer to the expired IOB, so that it can effect error recovery. The driver returns a code specifying whether the I/O was aborted or restarted, and whether a new time interval should be assigned.

The I/O subsystem makes a distinction between *solicited* and *unsolicited* interrupts. Device drivers are called with a code specifying whether an interrupt is considered solicited -- if there is active I/O on this controller, the interrupt is considered solicited. Device drivers then return whether the interrupt was handled.

If too many sequential, unsolicited, not-handled interrupts occur, the condition is logged, and that interrupt level is masked off. An operator can attempt recovery and bring the interrupt handling back on line.

Devices that normally generate unsolicited interrupts (such as for the receipt of a network packet) do not count these interrupts as unexpected interrupts.

Fatal I/O Errors

If a device driver or monitoring task detects too many consecutive I/O errors, it may log the condition and set devices or controllers off line.

Operators can also set devices on or off line.

The I/O subsystem checks that each I/O operation's buffer addresses and lengths are correct, that is, are in allocated buffer space before starting an I/O operation.

Device Drivers

Device drivers handle device and controller specific functions for the I/O subsystem.

To make device driver development as simple as possible, there is only one entry point in each device driver. The device or controller RDEV is passed a function code to describe the work to be performed, and possibly an IOB if this request pertains to a specific I/O operation.

As all RTOS and application code is shared, only one copy of the code for each driver exists in storage. Any specific instance of a device driver is known by the data structures passed to the driver.

A driver may be simultaneously called by multiple processors. For example, one processor may call a device driver to start an I/O operation, while another has called the device driver to handle an I/O interrupt. The RTOS only allows one processor to call a driver's interrupt service routine at a time.

It is the driver's responsibility to implement the minimum number of multiprocessing locks to successfully handle these conditions. The controller RDEV, and each device RDEV has lock bytes reserved for driver use.

Each RDEV reserves space for driver-specific data. These locations are normally used to point to driver-specific data structures.

Here is a representative list of function codes a driver may support:

```
enum {
   DEV_INIT,            // Initialize the driver
   DEV_IO,              // Start an I/O operation
   DEV_INT,             // Handle an expected interrupt
   DEV_UINT,            // Handle an unexpected interrupt
   DEV_MIH,             // Handle a missing interrupt
   DEV_HALT,            // Halt a specific I/O operation
   DEV_RESET,           // Reset a device
   BUS_RESET,           // Reset a bus
   CTLR_RESET,          // Reset the controller
   OP_SIZE,             // Return I/O unit requirements for this I/O
   POST_SCAN,           // Perform controller-dependent setup after
                        // a bus scan completes
   ENA_TARGET,          // Enable an emulated device
   DISA_TARGET,         // Disable an emulated device
   PREPEND_HEADER,      // Prepend physical layer header to user data
                        // (for Ethernet-type devices)
   DUMP_COUNTERS,       // Return statistical information
   ALLOC_BUFFER,        // Passes a buffer for the driver's use
   CLEAR_CHANNEL,       // Clear ALL active I/O operations
   DRIVER_POST          // Notify driver that a driver-requested
                        // delay time has expired
};
```

PCI bus scan

The RTOS system initialization scans each installed PCI bus searching for supported PCI devices. For each supported device found, the PCI scan creates a controller RDEV, then calls an instance of the device driver that supports this type of PCI device to reset the device/controller. This prevents any device from generating interrupts or engaging in other undesired activity until it is explicitly initialized.

After all busses are scanned, each device driver is again called to initialize the device/controller hardware.

The RTOS fills in PCI related information needed by the driver into the controller RDEV, such as the PCI address, vendor information, register and memory I/O addresses, and the interrupt number assigned to this controller.

During the PCI scan, any detected I/O memory is added to the RTOS translate tables to allow drivers to access I/O registers using memory instructions.

The system provides functions to allow drivers to read and write controller PCI configuration space.

Initialization

Each driver is responsible for initializing controller hardware, loading microcode (if necessary), setting appropriate initial conditions, and generally making the controller ready to receive I/O requests and handle interrupts.

The passed RDEV contains fields for the driver to store pointers to driver internal control blocks, and a set of lock locations to serialize processor access to driver resources.

The system configuration file may contain user-defined settings such as the SCSI ID's, FIFO sizes, buffer lengths, default bus speeds, number of devices supported, and so on. The RTOS collects these parameters into a common control block (*RCFG*) before the driver is first called. The driver can inspect this block to take relevant initialization actions.

Drivers can request special storage, such as contiguous regions larger than 4,096 bytes, specifically aligned storage, and storage with special caching requirements (such as cache disabled storage).

Interrupt queue support

At the end of driver initialization, the driver calls the kernel to specify which system interrupt(s) this controller uses.

The kernel queues this controller's RDEV into a linked list of controllers that process interrupt from this interrupt vector. When each interrupt for this vector occurs, this driver is called to (potentially) handle this interrupt. As some busses (for example, PCI) support multiple controllers on one interrupt vector, the driver should rapidly determine whether this interrupt is for its controller, and if not, return right away.

Since many interrupts can be handled by any of a number of processors, the I/O subsystem insures that a given instance of a device driver can only be active on a single processor. It **is** possible for one processor to be handling an interrupt while another processor is attempting to start an I/O operation. Or, several processors may attempt to start I/O operations simultaneously on the same controller or device.. It is the responsibility of the device driver developer to recognize these cases, and employ judicious use of multiprocessing locks to avoid driver reentrancy problems.

If a hardware platform supports multiple I/O interrupts, and if it provides a mechanism for prioritizing these interrupts, the RTOS can be configured to map physical I/O interrupts into a list of prioritized logical interrupts. The RTOS has no architectural limit on the number of I/O interrupt vectors supported – the Intel implementation supports thirty two I/O interrupt vectors.

I/O Operations

For each I/O request, the driver is called twice – once to determine the *cost* of the I/O operation in I/O units, then again to actually start the I/O operation.

The driver is responsible for associating the passed IOB with the actual physical operation started on the device or controller. To aid in this, a field is reserved in the IOB for driver use. Generally, the driver maintains its own control blocks for active I/O -- these may be in controller memory. The driver places a pointer to the IOB in these blocks.

In most cases, the buffer/length pointers or pointer (*SG*) list is of a form that can be directly used by the driver and I/O hardware. If not, the driver makes a local copy, or temporarily converts the passed list.

Here are some examples of the return codes that a driver can return to the I/O subsystem:

```
enum {
   IOS_STARTED,                // I/O started
   IOS_QUEUED,                 // I/O not started, must be requeued
                               // on the controller
   IOS_BUSY,                   // Controller unable to start the
                               // operation, must be requeued
   IOS_OFFLINE,                // Controller or device off line
   IOS_UNKNOWN,                // I/O subsystem received IOB with
                               // an unknown operation type
   IOS_BADIOB,                 // Bad IOB passed
   IOS_NOSTOR,                 // Insufficient storage to start
                               // the operation, must be requeued
   IOS_UNSUPPORTED_REQUEST,    // Unsupported request for this controller
   IOS_UNSUPPORTED_COMMAND,    // Unsupported command for this controller
   IOS_UNSUPPORTED_LENGTH,     // Invalid buffer length
   IOS_UNSUPPORTED_LBA,        // Invalid block address
   IOS_TIMED_OUT,              // I/O never finished
   IOS_START_ERROR,            // I/O never started
   IOS_ABORTED,                // I/O aborted
   IOS_IMMEDIATE_COMPLETION,   // I/O completed immediately, no need
                               // to wait for I/O interrupt
   IOS_NO_DRIVER               // Device driver does not exist, or isn't
                               // initialized
};
```

I/O interrupt handling

When a processor hardware interrupt occurs, the I/O subsystem inspects the linked list of controller RDEV's that process this interrupt, and passes control to each that has active I/O in process.

Each driver inspects its hardware registers to determine what (if any) operations have completed. These are posted so that waiting tasks can continue. Then, controller hardware-specific interrupt conditions are checked (an interrupt may be also be due to a controller error, a change in bus state, and so on).

Finally, the driver calls the I/O subsystem to propagate I/O request(s) from device to controller queues if necessary, and start queued I/O requests if necessary. This means that, in many cases, the driver's I/O start routine will be implicitly called by the I/O subsystem while the driver is within its I/O interrupt handler.

Unsolicited interrupt handling

Some devices, by their nature, will cause unsolicited interrupts, for example, an Ethernet controller will receive unsolicited interrupts whenever an incoming data packet is received.

Generally, any driver that needs to pass received data or requests to tasks or the system will have a head of queue list or a task pointer in their RDEV indicating where these data or requests should be posted (if data are received before the queue is initialized, the data are discarded).

For TCP/IP packets, the device driver queues packets in a common format to server task(s) responsible for handling that data. Some drivers may call fast path code to minimize processing for well known packets.

Device simulation

The RTOS ASC application supports the ability to emulate SCSI targets connected to other systems (hosts) through parallel SCSI and Fibre Channel connections.

To do this, device drivers must be able to collect an incoming SCSI Command Descriptor Block (*CDB*) and pass it to the application for analysis and execution.

As this is a special case of unsolicited interrupts that must be handled in a real-time fashion, the ACS application adds a kernel mode routine which is called by the driver to determine whether the request can be handled immediately (by passing back data buffers, good status, or exception status), or by instructing the driver to disconnect from the bus.

In the case of a disconnect, application code is dispatched to gather the resources necessary to continue the host request and issue an I/O operation to reconnect to the device and transfer data, or present *ending status* to the host.

Specifics of this device emulation application can be found in *The Design of an Advanced Storage Controller* by this author.

Driver Control and Debugging

Each developer of a device driver is actively encouraged to add console commands to assist in driver and hardware-dependent debugging and control. Typically these commands display hardware registers, internal statistics, perform selective resetting, and so on.

The RTOS provides generic commands to display and alter storage, I/O memory, and I/O registers.

Existing Device Drivers

The RTOS supports serial ports, parallel ports, IDE (ATA) drives, Parallel SCSI drives (Ultra, LVD and differential), Fibre Channel drives, Ethernet and Fast Ethernet, and v.35 (for direct T1/E1/T3/E3 access).

The serial port drivers support very basic modem control (dial-in enable/disable and dial-out), and can be configured to act as system consoles by emulating a subset of VT100 protocol. The serial port drivers also support direct connection and control of a Uninterruptable Power Source (UPS).

The parallel SCSI and Fibre Channel SCSI drivers support both initiator and target modes.

Ethernet, Serial Interface, and TCP/IP

The RTOS includes a TCP/IP stack supporting:

- Address Resolution Protocol (*ARP*).
- Internet Protocol (*IP*).
- User Datagram Protocol (*UDP*).
- Transmission Control Protocol (*TCP*) , including TCP for Transactions (*T/TCP*).
- Internet Control Message Protocol (*ICMP*).
- Domain Name Service (*DNS*) client.
- Dynamic Host Configuration Protocol (*DHCP*) client.
- File Transfer Protocol (*FTP*) server.
- Remote terminal protocol (*Telnet*) server.
- Hypertext Transfer Protocol (*HTTP*) server.
- Simple Network Management Protocol (*SNMP*) agent.
- Application-specific TCP and UDP messages.

Store-and-forward IP Routing

The RTOS can be configured to route specific and broadcast IP messages through to other physical interfaces, effectively acting as an IP bridge or router. This feature is not currently dynamically configurable.

Multi-Port and MultiHomed Host Support

The system can act as a multi-homed host, supporting multiple physical connections with either the same or different IP addresses. If multiple paths exist, network traffic is balanced among the paths. If one configured path fails, other configured paths are automatically used.

The RTOS can be configured to restrict access from specific IP addresses to specific sets of physical ports. This provides two benefits:

1. It can be used as a *firewall* to prevent IP *spoofing* (someone pretending to be at some else's IP address).

2. It can be used as a performance tool to balance physical line loads.

At a minimum, remote access to RTOS services are authorized by source and destination IP address. For some services, a user ID and password are required. Functions within a service may be restricted by user.

TCP/IP Application Support

The RTOS TCP/IP application interface provides application function calls (as shown below). The TCP stack is implemented by RTOS system tasks – consequently each connection or *socket* runs in parallel with all other connections. TCP support is multithreaded and takes advantage of multiprocessing.

To improve performance, there are two unique features of our implementation:

1. All transmitted data, and virtually all received data are transmitted from original data buffers. This means that there are no data copying between buffers as one goes from one protocol layer to another. This minimizes CPU, memory, and system bus utilization.

   There is no impact to the sending application as TCP, UDP, and IP headers are *prepended* before transmit using scatter-gather (*SG*) lists. The receiving application is passed entire received packets (*REMOTEMSG*'s), containing a pointer to and length of the *payload* (packets without data, and control packets are handled by the RTOS).

2. The application is responsible for determining when to send TCP acknowledgements (*ACK*'s). This places some burden on the application, but provides the application with the flexibility to optimize communications. The system handles retransmissions, opens, closes, and so forth.

   Applications using standard IP protocols, such as Telnet and FTP do not have to handle TCP ACK's.

Here is a sampling of application TCP/IP functions:

```
char sendudpmsg(struct BASE *, struct IPCONFIG *, unsigned long int,
   unsigned short int, unsigned short int, char *, unsigned short int);
                    // Send UPD message - parameters are base,
                    // ipconfig, dest. IP addr, source port,
                    // destination port, buffer address and length
char sendbroadcastudpmsg (struct BASE *, struct RDEV *,
   unsigned long int, unsigned long int, unsigned short int,
   unsigned short int, char *, unsigned short int);
                    // Broadcast UPD message down a specific path -
                    // Parameters are base, rdev, source IP addr
                    // destination IP addr, source port,
                    // destination port, buffer address and length

char tcpopen (struct BASE *, unsigned long int, unsigned short int,
   unsigned short int, unsigned char);
                    // Open a TCP connection - parameters are base,
                    // destination IP address, source port, destination
                    // port and open type
char tcpactivate (struct BASE *, unsigned long int, unsigned short int,
   unsigned short int);
                    // "Activate" a passive open TCP connection
                    // parameters are base, destination IP address,
                    // source port, dest, port
char tcpsend (struct BASE *, struct IOB *);
                    // Send TCP message, wait for ack
char tcpsendnw (struct BASE *, struct IOB *);
                    // Send TCP message, DON'T wait for ack
void acktcpmsg (struct BASE *, struct REMOTEMSG *);
                    // Send zero length ACK of received message
```

```
char tcpwait (struct TCPB *, unsigned long int);
                   // Wait for window size passed in parm 2
struct REMOTEMSG * tcpreceive (struct BASE *, struct TCPB *);
                   // Get or wait for next incoming TCP packet
unsigned long int gettcpsendwindow (unsigned long int,
   unsigned short int, unsigned long int, unsigned short int);
                   // Given source and destination IP addresses
                   // and ports, return number of bytes we can still
                   // send based on the window size and the
                   // sequence number pointers
unsigned short int getephport (struct BASE *);
                   // Allocate system-wide ephermial port number
char tcpclose (struct BASE *, unsigned long int, unsigned short int,
   unsigned short int);
                   // Close TCP connection - parameters are base,
                   // destination IP address, source port, dest port
char tcpabort (struct BASE *, struct TCPB *);
                   // Task immediately abort TCP/IP connection

char sendicmpmsg (struct BASE *, struct IPCONFIG *, unsigned long int,
   unsigned char, unsigned char, unsigned char,
   unsigned short int, unsigned short int, char *, unsigned short int);
                   // Send ICMP message - parameters are base,
                   // ipconfig, dest. IP addr, TTL, ICMP type, ICMP code,
                   // ICMP ID, ICMP seq. no., buffer address and length

char querypath (struct BASE *, unsigned long int, unsigned short int);
                   // Check whether a path to another system exists.
                   // Parameters are base, IP address, and protocol
unsigned long int netdoaping (struct BASE *, unsigned long int,
   long int *, long int *);
                   // Ping another system.  Return TTL and, optionally,
                   // the number of hops to send the ping request, and
                   // the number of hops to receive the ping reply

char cvtc2ip (char *, char, unsigned long int *);
                   // Convert character string into IP address
char cvtip2c (unsigned long int, char *);
                   // Convert IP address into character string
unsigned long int name2ip (struct BASE *, char *, char);
                   // Given system name and length
                   // return IP address or zero
char ip2name (struct BASE *, unsigned long int, char *);
                   // Given IP address fill in system name and
                   // return name length or zero
long utc2400 (void);   // Returns milliseconds since midnight UTC
long utc010100 (void); // Returns seconds since midnight, January 1, 1990
UTC short char2base64(unsigned char *, short, unsigned char *);
                   // Convert binary character string to BASE64 format
                   // Note: OUTPUT is 1/3rd BIGGER than the input
                   // Params are input address, input size, output addr
                   // Returns number of bytes in output string
short base642char(unsigned char *, unsigned char *);
                   // Convert BASE64 string to binary character format
                   // Note: OUTPUT is 1/3rd SMALLER than the input
                   // Parameters are input address, and output address
                   // Returns number of bytes in output string, zero
                   // if no string, or invalid character detected
```

And here are some of the TCP function return codes:

```
enum { NO_PATH = 64,             // No path to the remote system
       NO_TCPB,                  // No TCP block exists
       NO_IPCONFIG,              // System not configured to communicate
                                 // with this remote system
       TCP_NOT_ESTABLISHED,      // Connection not established
       ALREADY_ACKED,            // No need to send this ACK,
                                 // it is known that the remote system
                                 // has already received an ACK
       TCPIOBQUEUED,             // Packet queued rather than sent
```

```
                                // (window closed, or resources
                                // not available)
        CONNECTION_ABORTED,     // Connection aborted
        CONNECTION_CLOSED,      // Connection closed
        ALREADY_ESTABLISHED,    // Connection already established
        ALREADY_IN_OPEN,        // Connection already in open
        ALREADY_IN_CLOSE,       // Connection already in close
        CONNECTION_RESET,       // Connection reset by other side
        NOT_IN_PASSIVE_OPEN,    // Connection not in passive open
                                // (request to go from passive to
                                // active open)
        PASSIVE_TO_ACTIVE,      // Passive open became active open
                                // due to action from a remote system
        TCPWAIT_RECEIVED_ACK    // Remote system closed window is
                                // now open
};
```

Some of the RTOS TCP/IP operator commands include:

`PING`	Send an ICMP echo request to another system, display the reply.
`NSLOOKUP`	Given a domain name,. return an IP address, given an IP address, return a name. Multiple DNS servers are supported.
`TRACERT`	Show paths from one system to another (using ICMP echo's of increasingly larger IP *time-to-live* values).
`ARP`	Show physical layer connections and statistics.
`ROUTE`	Show routing table entries and statistics.
`RMTBLK`	Show system-to-system connections and statistics.
`SNMPDISP`	Display contents of one or all SNMP variables.
`TCPB`	Show TCP connections and statistics.
`USERS`	Show active Telnet, FTP, and HTTP users.
`DEFTCP`	Dynamically define this system's IP address.
`DEFUSER`	Dynamically add or delete a Telnet, FTP, and/or HTTP user.

FTP, Telnet, and HTTP

RTOS provides FTP, Telnet, and HTTP servers.

The Telnet server emulates ASCII and VT100 terminals.

The FTP server, in addition to allowing upload and download of files into the file system, supports a set of *simulated* files. These simulated files are used to upload and download data into flash memory and to hard disk system and configuration slots, for example:

CONFIG	Download the active configuration file.
CONFIG.*n*	Upload/download flash memory configuration file image 0-*n*.
DISKCONFIG.*n*	Upload/download hard disk configuration file 0-*n*.
DISKSYSTEM.*n*	Upload/download hard disk RTOS image 0-*n*.
DUMP.*n*	Upload system abend dump *n*.
LOG	Upload currently active system log.
LOG.*n*	Upload system log file *n*.
SYSTEM.*n*	Upload/download flash memory RTOS image 0-*n*.

Users can be specifically allowed or prevented from accessing each service, as well as portions of that service. For example, an individual FTP user can be prevented from uploading **anything**. Userid, password, and source IP address uniquely define users.

SNMP

The RTOS provides routines to enable the system and applications to act as an SNMP Version 1 agent

For multiple SNMP managers, the RTOS can be individually configured to:

1. .Receive and reply to SNMP variable requests.

2. Receive and accept SNMP variable updates.

3. Send SNMP traps on unusual or error conditions.

The vendor-specific SNMP variables and traps fall into three categories:

1. Management Information Block Version 2 (MIB-2) variables.. These include the variables required by any SNMP implementation, for example, system uptime, system location, system contact, etc.

2. System-specific. These include the RTOS version, storage size, active configuration file, overall statistics, and so on.

3. Application specific.

More information on these variables can be found in the *Design of an Advanced Storage Controller*.

File System

The beginnings of a file system are in place. Work in this area is progressing. These are some of the goals for the file system development:

- Fixed contiguous or dynamically allocated files.
- Update in place or reassign disk blocks.
- Ability to pre-allocate empty files.
- Long (64 byte) file names, (32 byte) file types.
- Fixed, variable, and stream record types.
- Subdirectories.
- Sharing among tasks and applications.
- Extensive permissions.
- Creation, last access, and last modification dates and times.
- Area in file directory entry reserved for application use.

System Commands and Privileges

System commands can be issued from the system console, from dial-in serial ports, or from Telnet sessions.

In all cases, there is a four level hierarchy of command privilege, the default being the lowest level:

None	Allow *display* commands only
Operator/Privileged	Allow *run-time control* commands
Administrator	Allow *configuration* and *performance tuning* commands
Maintenance	Allow *maintenance, diagnostic,* and *debugging* commands

User chosen passwords are required to enter all but the lowest privilege level. Each higher level enables all lower level commands. If an unauthorized command is attempted, an unknown command message is returned. The `HELP` command only displays commands authorized at the current privilege level.

Serial port dial-up access must be explicitly enabled by a system console command or via the system configuration file. Telnet access must be granted by defining user accounts.

These permissions only allow access to the system. Command privilege must be authorized separately by a `SET CMDAUTH` command, optionally prompting for a masked password.

The following RTOS commands are associated with serial dial-up support:

`ANSWER`	Answer an incoming call.
`DIAL`	Dial out to a user.
`HANGUP`	Sever a call.

Remote System Access

A system can be configured to allow remote operators to issue commands, receive command output, send messages to and receive replies from remote operators. These commands and messages are sent between systems using the UDP IP protocol:

`CMD system command`	Issue *command* at *system*.
`MSG system message`	Display *message* at *system.*

Remote commands are logged at the system executing the command.

Other Commands and Function Keys

A few other useful commands, not referred to elsewhere in this paper:

`CLS`	Clear the display output area.
`DATE/TIME`	Display the date and time.
`DO n command`	Execute a command *n* times concurrently.
`HT`	Flush pending, and stop displaying command output.
`EXIT/LOGOFF/QUIT`	Exit Telnet session (there just **can't** be enough ways to do this).
`REPEAT n command`	Execute a command *n* times sequentially.
`RT`	Resume displaying command output.
`RUNON n command`	Execute a command on processor *n*.
`SLEEP/DELAY n`	Pause for a period before processing the next command.
* (asterisk)	A comment line.

Key	**Function**
F12	Clear the screen.
Alt-F1	Lock the keyboard.
Alt-F12	Unlock the keyboard.

Abnormal Condition Handling

Any task or the kernel can request that the system *abend*. These *fatal breakpoints* are inserted by developers to fail on serious errors, or when debugging.

Based on the setting, an abended system can optionally dump all storage to a disk file, then either stop (for a debugging session), or automatically restart. The code and read/only data are checksummed when the system starts. If the checksum is still good, the current code is used, otherwise a new copy is loaded from the original flash or disk image.

Tasks can be started and stopped by operator command. Task information (such as the current instruction pointer, register contents, and stack subroutine trace-back) can be displayed. Tasks can also stop themselves using debugging checkpoints.

Debugging and Development Aids

Device Testing

Several commands exist to test attached devices in either a read/only or read/write fashion. These can be used to check hardware functionality or to exercise devices and the system. **As you will see from the list, many of these are very dangerous and are provided at the maintenance command privilege only**.

Some of these include:

`CCOPY`	Compare the data contents of two physical devices
`CDIO`	Test CD/ROM drives.
`CTLINIT`	(Re)initialize a specific controller.
`DBLOCK`	Display the contents of a block of data on a device.
`DCOPY`	Copy data blocks from one device to another.
`DEVNAME`	Given a device name, return the device address.
`DEVSCAN`	(Re)initialize a specific device
`DISKIO`	Test SCSI and IDE drives (format, drive buffer I/O, random, sequential, and repeated read, pattern write/read/compare, start/stop, mode and log select/sense, etc.).
`DRDEV`	Display status of one or more devices.
`FORCEINT`	Force the system to service a “pretend” hardware interrupt.
`HWPING`	Send one or more packets down a specific physical path.
`KILLDISK`	Stop running CDIO, DISKIO, and HWPING commands.
`OFFLINE`	Set a device offline.
`ONLINE`	Set a device online.
`PCI`	Show PCI configuration data
`PCISTORE`	Store into PCI configuration data.
`PORT`	Read/write values to/from I/O registers
`SCAN`	(Re)scan a SCSI bus.

See the *Design of an Advanced Storage Controller* for additional information on many of these commands.

Eye-Catchers

Each data structure has a four-byte *eye-catcher* to identify its use. This eye-catcher is comprised of printable characters identifying its current (or former use). For example, an I/O block might have an eye-catcher of *IOB* , and a released I/O block might have an eye-catcher of *FIOB.*

Eye-catchers are **never** used by software to identify a block, but are used by software to **verify the identity** of a passed block. Obviously, eye-catchers are very useful when debugging.

Symbol Table

As part of the compilation and linking process, the *symbol table* (function *entry point name* and *storage address* map) listing generated by the linker is converted into a module containing each name and storage address in storage address order.

The link is then reissued to include this symbol table into the storage image of the operating system.

The data in this table are used to convert code addresses into an *entry point plus offset* text string. RTOS commands are used to query this data, to display these symbolic names and offsets in stack trace-back displays, and to show the location of the failing instruction on system abends.

Trace Table

A single, time-stamped-entry trace table exists in storage to trace significant system events. Entry types can be disabled or enabled depending on the level of debugging in process. The size of the trace table can be changed by a compilation option.

For performance reasons, most of these entries are disabled by default.

Here are some of the events that can be traced:

- I/O requests.
- Hardware interrupts.
- I/O interrupts.
- Missing interrupt conditions.
- Locks and unlocks.
- Dispatches.
- Wait/Post.
- Transmission and receipt of TCP/IP packets.
- Storage allocation/release.

Certain function contain conditionally compiled debugging and informational message code that is disabled by default. These codes can be temporarily added to a debugging version to assist in isolating a problem.

Instruction Profiling

The RTOS can collect code usage counters by sampling the instruction pointer at each timer interrupt. A four-byte counter is maintained for each 32 bytes of code. A command is provided to display or reset these counters. A lower threshold can be specified so that only the most frequent values are displayed. The symbol table is used to display locations by entry point plus offset.

Debug Mode

The system can be directed to stop on an unexpected or requested crash. In this *debugging mode* the console can be used to inspect storage, device registers, control blocks, and so on. The console displays the date and time of the crash, the failing processor's registers, and the last 48 bytes of the stack pointed to by the stack pointer. All processors other than the boot processor are reset.

A large number of system commands are available for use in this mode (inappropriate or unusable commands are not allowed). Screen data and command output can be directed to a locally attached parallel interface printer. The active system image and/or configuration file can be changed (to back-off from new code, or to load a corrected configuration file).

Once debugging is completed the system can be restarted (using the currently loaded code), or booted (loading replacement code).

System Dumps

The RTOS can snapshot memory and hardware registers into one of a number of reserved areas on an internal hard drive (if installed and configured). Following the dump, the system restarts if possible.

These dump data sets can be used to identify problems after the fact, allowing the system to continue doing productive work while debugging proceeds.

Consistency Checks

There are a significant number of consistency checks within the system for both the kernel code and tasks/applications. These include:

- Eye-catcher verification checks of passed control blocks.
- Boundary alignment checks of released storage blocks.
- Size checks of released storage blocks.
- Queuing a block already queued to the same or another list.
- Attempting to remove an unqueued block from a list, or a block queued on a different list.
- Buffer address and length checks for I/O buffers.
- Checking that released control blocks do not have any subordinate control blocks queued off of them.
- Range checks for passed values.
- Using an already active IOB in a new I/O operation.
- Queuing work to tasks that aren't setup to be server tasks.

This list is by no means exhaustive. At last count there were over 500 separate checks within the system. Others can be conditionally compiled in for debugging thorny problems.

Performance Analysis

Many counters and statistics are maintained to assist in evaluating and tuning system performance. Here are a sample of commands used to display and alter performance:

`CACHE`	Shows storage utilization.
`CPU`	Shows processor utilization.
`INTCOUNT`	Display interrupt counters and dispatcher statistics.
`IOCOUNT`	Shows I/O statistics by device, controller, or aggregated by device type.
`PROFILE`	Report on sampling of instruction usage.
`QUERY`	Display current performance settings.
`SET`	Set a number of performance tuning knobs.
`TASKWAIT`	Display overall task statistics.
`UPTIME`	Show how long the system has been running.
`VERSION`	Show the RTOS version, build date.

System Management

Here are some commands used to manage the system:

`CONFIG`	Display configuration slots, change active RTOS and/or configuration at next boot.
`DELUSER`	Remove a user definition.
`DWARK`	Write currently running RTOS to hard disk slot.
`DWCONF`	Write active configuration to hard disk slot.
`FORCE`	Force a Telnet, FTP… user from the system.
`FWARK`	Write currently running RTOS to flash memory slot.

`FWCONF`	Write active configuration to flash memory slot.

Debugging Commands

The RTOS and ASC provide a large number of commands to aid in hardware and software debugging. These fall into two categories:

- General debugging commands – these are used for platform-independent debugging (such as storage display, module trace-back, control block chains, etc.) Here are some of them:

`DCONFIG`	Display the active configuration file.
`DISPLAY`	Display memory in a number of formats, display control blocks, trace table entries.
`DLE`	Display memory "the other way" (byte munged/flipped).
`DREAL`	Display the contents of *real* (versus *virtual*) memory.
`IOB`	Display status of active and queued I/O operations.
`MAPA`	Given an address, return the nearest entry point and offset from that entry point.
`SCREENQ`	Show queued output display messages.
`STORE`	Alter memory of a given size at a given location.
`STORSCAN`	Given an address, shows storage user and use.
`TB`	Show list of all, active, or waiting tasks.
`TDB`	Display all active time delay blocks.
`WFB`	Display all active wait/post blocks.

- Platform-specific debugging commands – these are used to debug driver or hardware platform-specific code. Commands in this category typically display hardware registers or platform specific data structures (some others are listed in *Device Testing* above).

`DREGS`	Show each processor's registers at the last system abend.
`MPI`	Display I/O APIC registers.
`MPL`	Display the local processor's APIC registers.
`MSR`	Display the local processor's model-specific registers.
`PAGTAB`	Display page directory and page table entries for the page at this address.
`PCIAD`	Display register and I/O memory addresses assigned to each PCI device.
`PCIIP`	Display interrupt number assigned to PCI device.

"Missing" RTOS Functions

Here is (an obviously very) incomplete list of some of the features the RTOS does **not** have (by design):

- No Graphical User Interface (*GUI*) – we plan to leverage the GUI on other systems by using their web browsers. If an application arises where local GUI support **is** required (for example, if the RTOS were to be used in a home application such as for internet support integrated into cable TV boxes), we would most likely port or develop sufficient GUI to support a licensed web browser.

- No dynamic load modules – as this system was developed to support a single real-time application, and as the hardware platform is not storage constrained, dynamically loadable code was not required.

- Neither multiple address spaces nor paging – the real-time applications envisioned for RTOS did not justify the overhead of address spaces.

- No interpretive command language such as REXX or Pearl – we hope to license and port an implementation of these languages. We did not have the resources or the immediate need to take on these developments internally.

- Source code level debugging – it was just too much trouble to work out another vendor's internal debugging formats, and didn't seem necessary (to me at least, several of our developers vehemently disagree).

Performance

It is relatively easy to collect performance data. It is often much harder to objectively compare data on disparate operating systems, hardware platforms, and environments. If you are running known benchmarks under the same operating system at the same release level on different hardware platforms, you can reasonably draw conclusions from the results (the I/O subsystem is faster, the processor is faster, there is more memory and less paging, more busses, faster devices, and so on).

When measuring a new operating system and non-standard applications, you clearly can't run the same benchmarks (not without modifying them anyway), and you probably are looking to measure different criteria. For example, measuring millions of floating point operations per second is beneficial if you are evaluating scientific computing environments, but largely irrelevant when comparing real-time systems.

So, when we talk of performance, we need to at least define *performance of what*?

In our case, we developed an application that behaves as a high-end advanced function storage controller that among other things:

- Attaches to unmodified hosts and servers via industry standard parallel and Fibre SCSI cables, simulating SCSI devices.

- Attaches to parallel SCSI and Fibre channel devices.

- Maintains a large internal cache for user data, supporting intelligent pre-fetch and destage of data

- Communicates among like systems via TCP/IP using high speed Ethernet cables.

- Transparently locally and remotely mirrors data to enhance overall facility reliability.

Running this application in the "real world" we have measured:

- Sustained data transfer rates of over 28 megabytes per second on each parallel SCSI interface.

- Sustained data transfer rates of over 55 megabytes per second on each LVD SCSI interface.

- Sustained data transfer rates of 100 megabytes per second on each Fibre Channel interface.

- Sustained data transfer rates of about 9 megabytes per High Speed (100 megabits/second) Ethernet interface.

- Host I/O rates in excess of 6,000 operations per second per interface.

Development Methodology

Here are some of the methodologies we employed to develop this software:

- A single overall RTOS architect – design input was always encouraged, but we had no design committees, formal reviews, and so on. I assume this method would be unrealistic with a larger or distributed development group.

- Evolutionary design – there may be people in the world who can completely design the perfect system, and anticipate every possible contingency and error scenario ahead of time. I, unfortunately, am not one of them. So, as often as we felt the urge to stick to a design we invested time and effort into, we would go back and tune, improve, change, or revamp portions of code as we gained real-world experience.

  Also, the world changes out from under you. What made sense four years ago, doesn't necessarily make sense four years later (this is also unfortunately true of market requirements).

- One person, one function -- generally, each developer was fully responsible for one area of development. The scope of that function was defined by its complexity and the need for specific knowledge. For example, one person developed all of the parallel SCSI device driver code, another developed all of the Fibre Channel code, and another developed all of the Ethernet and serial interface code.

- Think first, code second – Without exception, any time something was developed in a rush, or in an ill-considered manner, the result was worse than useless – after spending inordinate time attempting to patch it into submission, it had to be redeveloped from scratch.

- Provide a comfortable, productive working environment – for us this meant large, high resolution screens, very fast workstations, one-person offices with real walls (**not** cubicles), very high speed networking and fast Internet access, individual choice of text editor, keyboard, and mouse. We also placed fully standalone test systems on rolling carts that could be moved into developer and tester offices. These systems could be linked with lab or other test systems via separate high speed networks (we wired at least three high speed Ethernet lines into each office). This, and FTP upload of code allowed for a very fast change and test cycle.

- Debug the smallest possible portions of code before moving on. You will no doubt find problems when integrating code, but is much easier to fix the easy problems in one portion of code at a time.

- Collect evidence, then read the code to understand what is wrong. Don't depend on the test group to test something into working.

- When you find a bug, assume the same bug may be repeated elsewhere, so look elsewhere in the code for similar problems.

- Developers can and should test, but should **never** develop their own regression tests. If a person misunderstands how something should work, the test will contain the same misunderstanding.

- Develop with compiler optimization on – we did this because this is the way we anticipated releasing the code, and didn't want to have to deal with a whole system full of last minute bugs introduced by timing or code reordering of the compiler.

- Copious comments. We've commented every line. We do this for self preservation – for me at least, it is hard to remember how and why I did something two weeks ago, let alone four years ago.

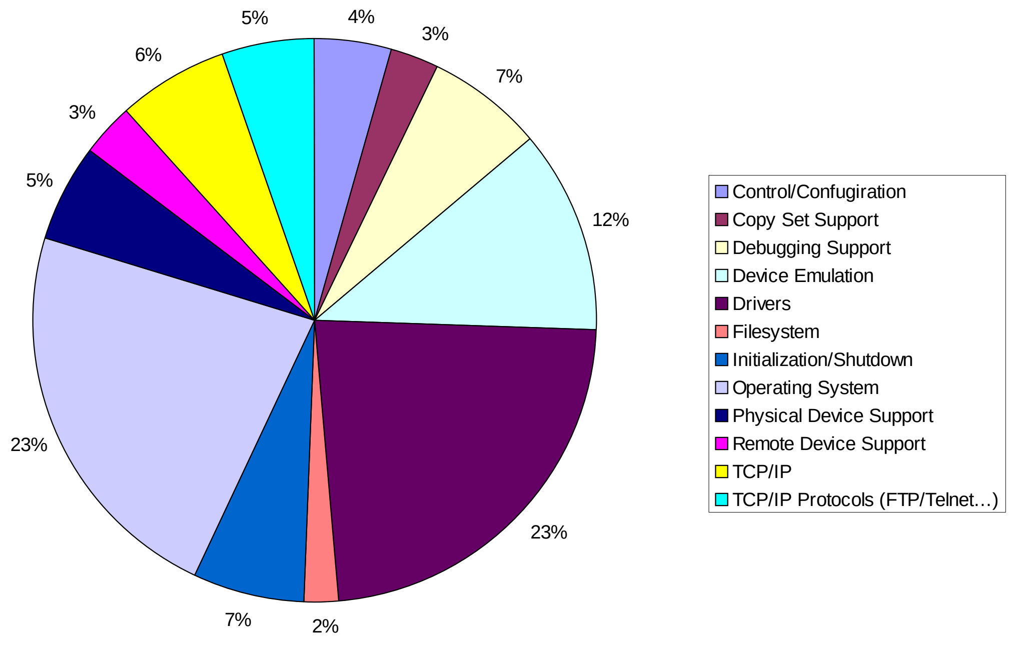
- Invest more on testing than on development. I once had a saying: "Anything not tested doesn't work." As I've gotten older and more experienced, I've changed this to: "Anything not tested **lately** doesn't work." Spare no expense on test equipment, including real-world environments. I have worked on many platforms from mainframes to minicomputers and have never seen a standalone test suite that eliminates the need to test with real hosts, servers, devices, applications, and databases.

Development History and Statistics

This section briefly shows some interesting development statistics, such as the percentage and time needed to develop each functional area of RTOS. This data were gathered in mid 1999.

Code Size By Function

The following graph shows code size by general function:



As the graph shows, the basic operating kernel accounts for about 24% of the total code, kernel support (initialization, shutdown, debugging commands, configuration) is about 16%, device drivers and device control about 30%, TCP/IP and TCP/IP utilities 12%, and the application code 18%.

This tells us that about one-fifth of this development effort was application-specific (rather than system and general-use) code..

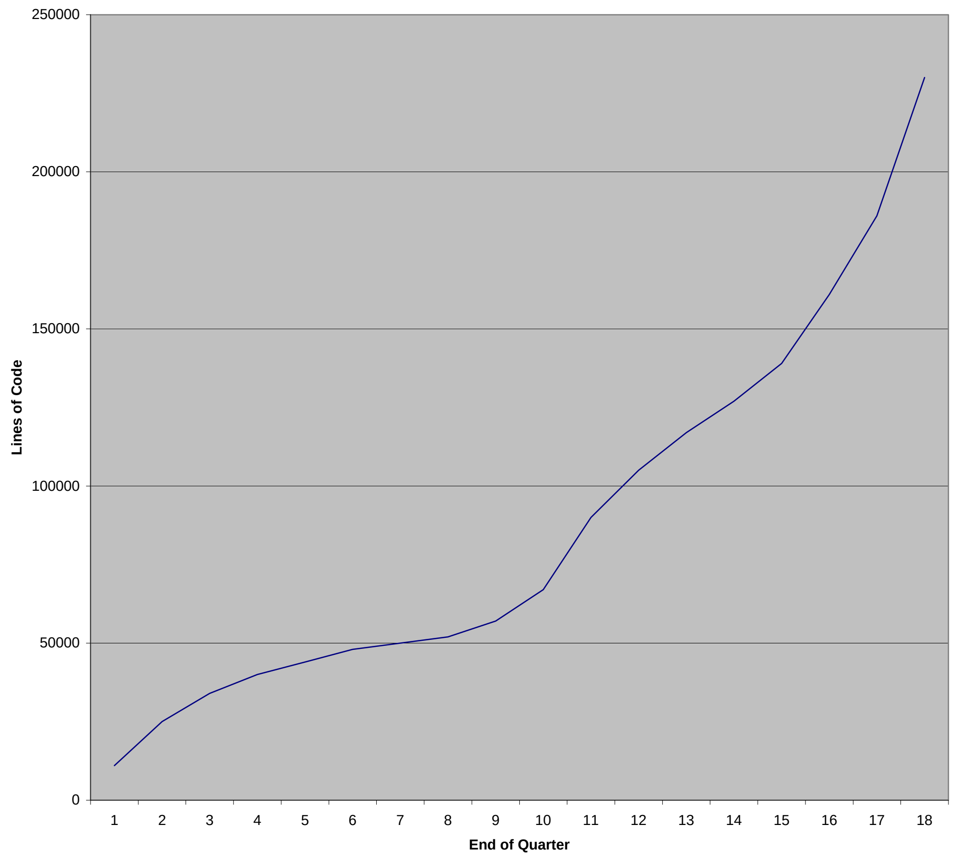
Device driver code accounts for almost one-third of the project. As the drivers we developed control very sophisticated controllers (with their own microprocessors), this is not entirely surprising.

Code Size Over Time

The following graph shows the size of RTOS and associated applications in lines of code over the roughly four years of development.

There are three roughly linear regions to this graph. The first was the period of part-time development "before outside funding.". The second portion shows a period when developers were also handling non-technical tasks. The final portion shows true dedicated development effort.

These data were gathered in mid 1999. As of this update (January, 2000) there are about 240 source files, and 241,000 lines of code.



Code Cost

This section gives a rough estimate of the amount of time each function took in *developer months*. Some facts should be kept in mind when reviewing these data:

- Not all developer months are equal. Some developers develop more code faster than others. Some developers cost more than others. Some code needs less debugging.

- Not all development efforts are equal. A device driver for a complex, poorly documented piece of hardware is significantly more difficult than, say, formatting and displaying console messages.

- As a project progresses, developers inevitably become more interrupt driven, as they probably maintain and extend old code as well as develop new code. This is exaggerated further when customer problem resolution is inevitably added to the effort.

As of mid 1999, approximately 125 developer months have been invested into this project. These are *wall clock* months, not actual development time months (which we did not endeavor to measure). The **average** number of debugged lines of code developed per wall clock month is approximately 2,000.

Function	Approximate developer months
Basic Operating System	12
Operating support (debugging, initialization, etc.)	12
TCP/IP	14
TCP utilities, FTP, Telnet, HTTP, skeletal web page designs	11
Parallel SCSI device drivers	30
Fibre Channel SCSI device drivers	12
Ethernet drivers	6
WAN driver	6
Device emulation	6
Remote device application	8
Device mirror set application	8

Summary

This paper presented the Ark Real Time System. It gave an overview of the system, design goals, and some of the development methodologies and productivity results.

References

<to be completed>

Acknowledgements

<to be completed>