# RIDGE OPERATING SYSTEM REFERENCE MANUAL

February 24, 1983

# SECTION 1

# OPERATING SYSTEM ARCHITECTURE

The Ridge Operating System (ROS) is a multiprocessing operating system for multiple users in a virtual memory environment, with extensive file system support, input/output device interfaces, and message-oriented interprocess communication facilities. ROS consists of a set of utility "commands" and a set of program-callable "interfaces" that together provide a foundation for application-level activities. The extensible architecture of ROS allows new commands and interfaces to be added by the user to the existing structure of the operating system.

The design of the Ridge Operating System has been heavily influenced by the UNIX operating system. The hierarchical file system, system interface routines, and utility commands of ROS are very similar to UNIX, thus providing a well-known environment to those users who are already familiar with UNIX or other similar systems.

The underlying architecture of ROS, however, differs from UNIX in several significant ways, which contribute to high performance, reliability, and extensibility. The operating system is composed of several independent processes that communicate via messages rather than sharing memory. Virtual memory management, the file system, and the interprocess communication mechanisms are completely integrated with each other, thus leading to high performance. The functional isolation of the components of the system leads to better reliability. The modularity of the system allows easy extension by adding new service processes that are independent of other system processes.

The majority of ROS is written in the programming language Pascal (refer to the Ridge Pascal Reference Manual). The system interfaces are described in terms of Pascal definitions throughout this manual. Other high-level languages provide their own interfaces to the system, which are described in the appropriate language manuals.

## KERNEL

The Kernel is the lowest level software in the operating system. The Kernel implements process creation and deletion, multitasking primitives, and message-oriented interprocess communication. Several memory management functions are handled by the Kernel, including maintenance of the virtual-to-real translation table, which supports the virtual memory system. The Kernel also provides the first level handling of program traps and faults, and dispatches input/output device drivers when an interrupt occurs.

The Kernel routines are the only routines in the system that run in a special processor mode called "kernel mode". Kernel mode is used to provide all privileged activity that involves data sharing or synchronization. In kernel mode, the processor uses real memory addresses instead of performing address translation of virtual addresses. Certain privileged processor instructions are valid only in kernel mode, thus allowing the Kernel to protect user and system processes from interfering with each other.

All other routines in the system run in "user mode" which uses virtual addresses. User and system processes request Kernel services via interface routines that cause the Kernel to be entered, thus switching to kernel mode. When the Kernel routine is completed, the processor returns to user mode and execution proceeds.

The Kernel is also entered whenever an "exception" such as a program fault or an external interrupt occurs. There are several mechanisms for dispatching a process as a result of various exceptions, thus allowing the process to handle the exception.

## Processes

Processes are software entities which perform some computational task, interacting with other processes to provide application-level functions or system services. A process is an instance of a running program. All user programs run as processes, and the operating system itself is composed of a set of processes that manage various system resources.

Each process has a set of private address spaces, including a code segment, a data segment, and a queue segment (used for interprocess communication). Each process also has a state maintained by the Kernel which includes the current program counter and the general registers. Processes share the use of the processor via priority scheduling, and thus appear to be running concurrently with each other, allowing multitasking and multiprocessing.

## Objects, Managers, and Clients

The hardware and software resources of the Ridge computer system are represented in ROS as "objects". Examples of hardware objects include the processor, memory, disc volumes, terminals and other input/output devices. Software objects include files, directories, and user programs.

Objects are controlled by processes called "managers". A manager is responsible for creating, deleting, and modifying objects under its control. Each manager specifies an interface that allows access to its objects. The interface specification defines the services of the manager, and allows the implementation of a manager to vary without impacting the users of its services. ROS includes managers for virtual memory, volumes, directories, files, terminals, and other input/output devices. Manager processes can be created to manage new types of hardware or software objects.

Application programs, on behalf of their users, become "clients" of the services of a manager. A manager may become a client of another manager which helps it accomplish its task. A client process requests the services of a manager via the interprocess communication primitives supported by the Kernel. Examples of clients include processes requesting access to files, directories, device drivers, or other user or system processes.

### Messages

Processes communicate with each other by exchanging "messages". At the lowest level, messages are fixed size blocks of data that can be sent or received via the Kernel. Processes use messages to synchronize their activities as well as to share data with each other. The message mechanisms are also used by the Kernel to indicate events such as external interrupts, faults, or traps encountered by a process.

Two processes that wish to communicate using the Kernel primitives must set up two unidirectional communication paths, one in each direction. A "link" is an outbound channel from one process to a queue of another process, while a "queue" is an inbound channel that may receive messages via one or more links. A message is sent on a particular link, and is buffered in the associated queue until the receiver requests the message. The link, queue, and message buffer data structures are maintained in the queue segment of each process, and can only be accessed by the Kernel.

Context switching from process to process is handled by the Kernel as a side effect of process communication primitives. Attempting to receive a message on an empty queue causes the receiver to be suspended until a message appears, allowing other processes to run. Sending a message may suspend the sender if the receiver is of higher priority and is waiting for a message.

The sequence of information that is exchanged between a client and a manager is called a "protocol". Each manager defines a protocol that is used to access objects under its control. A message-oriented protocol usually consists of a set of request messages and corresponding response messages.

Client processes usually provide some insulation from the details of manager-object protocols in the form of "runtime libraries". A runtime library contains subroutines that are linked with the user program, and among other things, handle the details of communicating with the system processes.

## FILE SYSTEM

The ROS file system supports the storage and retrieval of data on secondary storage devices (usually discs). Files serve as both long-term storage of data to be used by one or more programs, as well as backing store for the code, data or queue segments of the virtual address space of a process.

A "file" is a sequence of 8-bit bytes, with no other interpretation placed upon the data by the file system. A file may be given a name and cataloged in a "directory". Commands and interface routines exist to create, delete, read, and write files; other modes of access include listing directories and changing file properties.

The file system is a collection of manager processes, with each manager implementing a part of the overall services provided by the file system as a whole. The file system provides interfaces for accessing files via read and write requests from user and system processes. In addition, interfaces exist that allows virtual memory management to be integrated with the file system.

## Volume Manager

The Volume Manager maintains the space on a disc volume, allocating pages of 4096 bytes each to files. An internal file identifier is used to specify a file to the Volume Manager. The Volume Manager creates, deletes and changes the space allocation for a file, and maintains information about the file such as its size and the time of creation or last reference in a "file label".

The allocation of a file to actual disc addresses is handled by the Volume Manager using an extent-based data structure. An "extent" is a variable number of contiguous pages on the volume that corresponds to contiguous pages in the file. Sequential accesses can be thus be accomplished with minimal disc latency.

An extent can be identified simply by the address of the first page and the number of pages in the extent. All the extent identifiers for a file are kept in a single "extent page", which guarantees that at most one extra disc access would be necessary to locate a given page of a file. The Volume Manager attempts to allocate single-extent files if possible. A single-extent file does not use an extent page, thus eliminating any extra indirect access.

The Volume Manager interacts with the virtual memory system to map file accesses to actual disc addresses. The virtual memory system supports the Volume Manager by making its data segment be the entire disc volume. This permits the Volume Manager to easily manipulate file labels and extent pages by making references to the virtual addresses that correspond to the disc addresses of the desired file labels and extent pages. The most frequently referenced Volume Manager data will tend to stay in memory through the virtual memory system's page replacement algorithms, thus increasing file system performance. Multiple volume managers are used in the case of multiple disc volumes.

## Directory Manager

The Directory Manager maintains the mapping of character string file names into internal file identifiers. The internal file identifiers are used by the Volume Manager and the virtual memory system to specify a file object, rather than referring to it by the character string.

A basic name in the file system consists of from 1 to 16 characters. The characters may be either alphabetic ("A" to "Z", or "a" to "z"), numeric ("0" to "9"), or the period character ("."). The name must start with an alphabetic character. Upper and lower case alphabetic characters are stored and retreived as given, but upper case matches lower case and vice versa when a name is looked up.

By convention, the period character "." is used to separate a name into one or more parts, with the trailing parts called "extensions". For example, program source files usually end with the extension ".s", as in "prog.s". The number of extensions in a name is not limited by the file system; the file system itself places no interpretation on file name extensions.

The directory structure is hierarchical; directories may contain other directory names as well as file names, thus providing a tree-like organization. A "pathname" is a string of characters that uniquely identifies a file or directory. The sequence of directory names starting from the file system "root" describes a path within the directory structure that specifies a particular node in the tree.

The root directory is specified as "/"; the sequence of directory names are then concatenated together with a "/" separating each name, where left-to-right corresponds to higher-to-lower from the root towards the leaves in the tree structure. For example, "/usr/bin/date" would identify the "date" file within the "bin" subdirectory within the "usr" directory that springs from the root directory.

Each user has a "current working directory" which supplies the default pathname prefix for a name which does not start with "/". For example, if the current working directory is "/usr/bin", then the name "date" would be equivalent to the full pathname "/usr/bin/date". If the current working directory has subdirectories, the pathname of a file therein begins with the name of the subdirectory with no leading "/".

The current working directory can be changed to provide access to files in other directories using shorter, partially specified pathnames. A fully or partially specified pathname may be used anywhere a file name is required in user commands or as parameters to interface routines.

Two special directory names are recognized by the Directory Manager. The name "." in each directory stands for the directory itself. Thus the current working directory can be referred to as "." without having to know its complete pathname. The name ".." refers to the parent of a directory, that is, the directory that contains a given directory. These special names can be used with any directory and as a part of any pathname, but are not actually stored in the directories. Instead, they are interpreted by the Directory Manager as part of resolving pathnames.

The data structures defining the directory hierarchy are part of the data segment of the Directory Manager, which is allocated when the disc volume is initialized and is proportional in size to the disc volume. The directory data structures remain in a contiguous area of the disc, and the most frequently referenced Directory Manager data will tend to stay in memory through the virtual memory system's page replacement algorithms, thus increasing file system performance.

### File Manager

Each open file in the file system has a separate File Manager associated with it. The File Manager accepts requests to read or write blocks of data of up to 4096 bytes at a given page within the file.

A File Manager allows access to a file as though it was a variable-length, contiguous sequence of bytes, which insulates the client of a File Manager from the details of the extent-based disc space allocation. The size of a file grows automatically as new bytes are written to the end of the file. The desired page address is given as a parameter of each request, thus supporting sequential or random access to the file.

A File Manager process is created when a user process opens a file, with the file as the data segment of the File Manager. The user process makes read and write requests directly to the File Manager, which then references the requested page as part of its virtual address space. The File Manager process is deleted when the file is closed.

File pages are transferred to and from secondary storage as a part of the virtual memory system's management of memory. The entire main memory thus serves as a global buffer pool for file pages. A page of a file is moved directly from the disc volume to the user address space and vice versa via the Kernel message primitives. This is accomplished without any memory-to-memory copying required, thus increasing the speed of file access.

## VIRTUAL MEMORY MANAGER

The virtual memory system consists primarily of the Virtual Memory Manager, which interacts with the Kernel, the file system, and input/output processes to implement the virtual memory environment for all ROS processes. The Virtual Memory Manager maintains real memory pages of 4096 bytes each, assigning them to virtual pages for each process on a demand basis.

The Kernel maintains the Virtual to Real Translation (VRT) table, a table in memory which contains entries mapping virtual pages to real pages. The Virtual Memory Manager makes requests to the Kernel to examine and modify entries in the VRT, thus mapping virtual pages in secondary storage to real pages in memory.

When a process attempts to access a virtual page that is not mapped in the VRT, a "page fault" occurs and the Virtual Memory Manager is notified by the Kernel. Page faults are handled by the Virtual Memory Manager, which interacts with the Volume Manager and various peripheral device driver processes to find and then read or write the appropriate secondary storage to or from memory. Once the page has been mapped, the process can be resumed at the instruction that caused the page fault. If there is no virtual page assigned for a particular reference, then a message is sent to the parent of the faulting process which is responsible for handling the page fault.

The memory hardware maintains referenced and modified ("dirty") bits for each page. These bits enable the Virtual Memory Manager to implement sophisticated page replacement algorithms that tend to keep the most frequently referenced pages in memory, thus improving system performance.

## DRIVERS

A "device driver" is a software process that interfaces directly to a hardware peripheral device. A driver is responsible for initiating input and output commands to control a particular device, and servicing interrupts generated by that device. A driver also has a client interface which accepts requests from other processes to perform input and output on their behalf.

There are several Kernel interface routines to help support device drivers. A device driver can associate itself with a particular device, and then receive a message when that device generates an interrupt. Direct memory access (DMA) by input/output devices uses real memory addresses, not virtual addresses, so there is a routine that translates a virtual address to a real memory address.

There are Kernel routines to lock a page in memory, making it always resident, or to unlock a page. A page must be locked during access by a device to insure that its contents are not disturbed by the virtual memory system. Part of the queue segment for a device driver should be locked in memory so that interrupt messages are not discarded.

A device driver provides a client interface that allows its device to be accessed in the same manner as a regular file in the file system. That is, the same messages are sent to open, close, read, and write a device driver as a file. Additional device control protocol may be defined by a particular device driver, which serves as the manager of the device object.

Currently, the User Monitor process serves as the device driver for all devices accessible to commands and user processes. The User Monitor manages user processes, providing them an interface to the rest of the system, and is described later in this section. The User Monitor contains a fixed set of device handling routines for the terminals, printers, and floppy disc devices that can be attached to a Ridge system.

The names of the ROS-supplied device drivers always begin with a colon

":" to differentiate them from regular files in the file system, since a device driver name can be used anywhere a pathname is accepted. For example, the terminal is referred to as ":term". These names are recognized specially by the User Monitor in an open request, and further input/output requests to the device are handled by the User Monitor. The ROS device driver interface and naming conventions are subject to change.

## Terminals

The name ":term" refers to the device driver for a particular user's terminal. The terminal device may be either a Ridge display, or a CRT attached to the J-1 RS-232 connector (with a baud rate of 9600).

The terminal device driver buffers up to 256 incoming characters, allowing "type-ahead". The device driver ":term" supplies one input character at a time, which is echoed to the terminal when a program makes a read request to the driver. The device driver ":termnoecho" is similar, but does not echo its input.

The Control-C character is used to indicate end-of-file on input. There is currently no special handling of input characters such as Backspace or Return by the device drivers, although most programs do interpret these control characters in a conventional manner as described later.

On output, one character at a time is sent to the device; if an ASCII Return character is output, an ASCII Linefeed character is automatically appended.

## Printers

The name ":lp" refers to the device driver for the Data Products/Centronics compatible line printer. The name ":vers" refers to the device driver for the Versatec printer/plotter, which currently is only supported in character mode (not bit-oriented plot mode).

Both printers are output-only devices, which buffer characters to be printed until at least an entire line is accumulated. The device drivers accept single characters to be written at a time; if an ASCII Return character is output, an ASCII Linefeed character is automatically appended. When the device driver receives a close request, an ASCII Formfeed is output.

## Floppy Discs

ROS supports the floppy disc drive packaged in the Ridge system processor cabinet. If an additional floppy disc drive is attached, then the drive mounted in the processor cabinet is referred to as the "left" drive, and the other is the "right" drive.

The floppy disc can be either single or doubled-sided, with either single or double density formatting following the IBM 3740 soft-sectored standard. A floppy disc is normally formatted by ROS utilities as double-sided, double-density with 512 bytes of data per block.

The directory and file structure maintained on a floppy disc is compatible with the University of California, San Diego (UCSD) Pascal system format. This structure allows up to 77 files per volume, with a single directory for the entire volume. The name of a file on the volume may have from 1 to 15 characters, which may be either alphanumeric ("A" to "Z", "a" to "z", "0" to "9") or a period ("."). Upper and lower case match each other, but upper case is always stored in the directory.

A file on the floppy disc is referred to by its name, with the colon prefix ":". By default, floppy disc files are accessed on the left drive; the prefix ":L: or ":l:" explicitly accesses the left drive; if the prefix ":R:" or ":r:" is specified, then the right drive is accessed. For example, both ":data" and ":l:data" refer to a file whose name is "data" which is found on the left drive.

Files on a floppy disc may be read or written a block of bytes at a time, with no transformation or interpretation of the data performed by the device driver. Several ROS utilities are provided to format a new floppy disc, to list the directory, and to compact the space allocation of a floppy disc.

### Null Device

The name ":null" refers to the null device driver, which serves as a sink for unwanted data. Data written to the null device is always discarded, and no data is returned with an end-of-file status if the null device is read.

## USER MONITOR

Client processes (including user-written application programs) generally request operating system services via the User Monitor process. There is one User Monitor process per active user. The User Monitor receives requests from client processes and interacts with the appropriate object managers on behalf of the client.

The User Monitor helps to simplify the system interfaces by insulating the client processes from the details of interacting directly with the various object managers. In addition, much of the access control and other protection mechanisms supported by the system can be implemented by the User Monitor rather than by each manager.

A user program or a system-supplied command runs as a "user process" which is created and managed by the User Monitor. The user process communicates with the User Monitor to open and close files or device drivers, query and modify the file system, or create and manage other user processes. The User Monitor maintains an environment for each user process, which includes information about its code, data, and queue segments, its open files, and its command arguments.

When a program is run as a process, it will typically access one or more input or output files. The program opens a file by name via a system interface routine. The User Monitor handles the "open" request by binding the program to the appropriate File Manager, device driver or other input/output process, which then services "read" and "write" requests. When the program is finished with the file, a "close" request causes the file to become inactive.

A "name equation" provides a mapping from one string to another. The User Monitor maintains a set of name equations for each user process that can be used as convenient abbreviations for a pathname, or can be used to redirect a program's access to different named objects without changing the alias name used by the program.

Many programs use one file for input, one file for output, and may need to report errors to the user on another output file. The name equation mechanisms can be used to support this common need, allowing the standard input, standard output, and standard error output of a program to be redirected to different files each time the program is run.

For example, if a program always opens its standard input using the name "input", then setting the name equation for "input" to map to some file name before running the program will cause that file to be accessed, without requiring changes to the program. Likewise, by convention, the name "output" is used to refer to the standard output, and "stderr" is used to refer to the standard error output. The default name equations for these names, and any changes to the mappings, are typically handled by a command interpreter process which invokes user processes via the User Monitor.

## SHELL

The Shell is a command interpreter process that is used to invoke programs that execute as commands. The Shell interacts with a user at the terminal, reading text lines typed in and interpreting them as requests to execute other programs.

The Shell runs as a regular user process, getting its input either from the terminal or from a file containing commands. The first word in each line is taken to be the pathname of a program's code file, and any remaining arguments are either interpreted by the Shell or passed on to the command process.

The Shell makes requests to the User Monitor to invoke command processes and redirect their standard input or standard output. Command arguments are accumulated by the Shell and passed to the User Monitor, which then delivers them to the user process when they are requested. The Shell also expands file name patterns by reading file system directories and generating all the matching names for a given pattern.

The Shell and the command environment is described in detail later in this manual.

## RUNTIME LIBRARY ROUTINES

A user program interfaces to the system via routines that are linked into the program. A "runtime library" contains routines that handle the details of communicating with the system processes, and provide other common functions.

The ROS system interfaces described in this manual are referred to as the Runtime Library interface, which consists of Pascal-callable routines that may be referenced by a user program. Other runtime libraries may be created that support additional features and interfaces, for instance, to support other high level languages.

The Runtime Library supports a higher level abstraction for dealing with system objects such as files and devices. The Runtime Library routines implement "streams" on top of the fixed size block-oriented Kernel message primitives. A stream of data can be accessed a byte at a time, with the necessary buffering or device-dependent aspects of managing the data hidden by the routines.

The stream file interface provides a uniformity of access to the file system, device drivers, and other processes which fit the model of access defined by the file manipulation routines. For example, a program uses exactly the same routines to open, close, read and write a device driver that it uses for a file system access.

A calling program does not have to be concerned with the setup of interprocess communication channels, or other system implementation details such as the mapping of directory names to internal file identifiers, volume space allocation, or peripheral device initialization and servicing. Thus, the Runtime Library routines support a more general model of access for a "file" than that provided by the implemention of the actual file system or device drivers.

The ROS-supplied file "/lib/rtl.o" contains the Kernel interfaces and Runtime Library interfaces described in the other sections of this manual. This file should be linked with a user program that uses those interfaces. The user program may invoke the routines either directly or as a result of language-defined operations that are built on top of the basic interfaces.

## SYSTEM INITIALIZATION

The Ridge Operating System software resides on secondary storage, and during normal operation only active parts are resident in memory due to the operation of the virtual memory system.

## System Installation

The system software is installed on the secondary storage using a procedure described in the Ridge Operating System Installation Guide. This procedure is used to build a new file system, or modify an existing one, to include new or changed components of ROS supplied by Ridge Computers. The result of installing ROS will be a file system with a prototype directory structure that contains the code files of all the system processes and utility commands.

The directory "/ROS" contains the files for the Kernel, Virtual Memory Manager, Volume Manager, Directory Manager, File Manager, User Monitor, and other code, data and queue files used by the operating system.

The directory "/usr" is the typical place new user directories are created.

The directory "/bin" contains the most commonly used commands (in binary executable form), while "/usr/bin" contains commands used less often.

The directory "/lib" generally contains library routines to be linked with user programs.

## System Load

System initialization occurs when the processor is first powered on, or when the processor is reset by pressing the Load Switch. The operating system goes through a procedure called "bootstrapping" during which the necessary parts of ROS are read into memory from secondary storage.

Upon reset, the microcode of the processor sends a request to the bootstrap device selected by the Device Switch. The bootstrap device then reads a small block of code into memory, which contains the Bootstrap Debugger RBUG (refer to the Ridge Bootstrap Debugger Manual).

RBUG contains a table of disc addresses and sizes of possible files to be read into memory. By default, RBUG reads the file referred to by the first entry, which contains the Kernel, Virtual Memory Manager, and the Volume Manager code. Then RBUG begins executing the file just read in.

The Kernel is started, which initializes itself, and then creates the Virtual Memory Manager and Volume Manager processes.

The Virtual Memory Manager checks the current system date and time, requesting that a new date be entered if the date is nonsensical, as is the case when the system is first powered on. The input format for the date is given in the message at the terminal, and is similar to the "date" command described later in this manual.

The Volume Manager then finds the Directory Manager code file, which has a special internal file identifier known to the Volume Manager. The Directory Manager process is created. The Volume Manager also creates a process called Startup, which also has a special internal file identifier for its code file.

The Startup process finds the file that contains the User Monitor code by interacting with the Directory and Volume Managers to find "/ROS/um". Startup creates one User Monitor process for each Ridge display attached to the system. Each User Monitor uses a different display for its terminal. Currently, if no display hardware exists, then a single User Monitor is created and it uses the CRT attached to the J-1 RS-232 connector for its terminal.

Each User Monitor sets its current working directory to the root "/", and then starts a process using the file "/bin/sh", which normally is the Shell command interpreter. The Shell prints its prompt "$" to indicate its readiness to accept commands.

# SECTION 2

# COMMANDS

Commands are programs which are invoked directly by the interactive user of the Ridge Operating System. The user interacts with a command interpreter program, called the Shell, via a terminal. The Shell reads lines of text and interprets them as requests to execute each named command, which may be either a system-supplied utility or a user-written program.

## USING THE TERMINAL

Most programs, including the Shell, support a simple, standard interface to the terminal.

Type-ahead is possible, which means characters can be typed even while a program is printing characters on the terminal. Whatever is typed will be saved and interpreted in correct sequence. There is a generous limit on the amount of type-ahead characters; if the limit is exceeded, all the saved characters will be thrown away.

Input from the terminal is typically line-oriented, with typed characters echoed and accumulated until a RETURN character is typed. Typing mistakes can be corrected in the input. The backspace character (BACKSPACE or Control-H) erases the last character typed. Successive uses of backspace erase characters back to, but not beyond, the beginning of the input line.

The terminal may be used as a source of input wherever a file is expected. The Control-C character is used to indicate end-of-file. The input is terminated, and the Control-C character is not passed through to the requesting program.

The DEL character is used to abort programs, such as those that run indefinitely or produce unwanted printouts. All processes started by the user are terminated when the DEL character is typed, except for the original Shell created during system load.

## THE SHELL

The Shell is a command interpreter that reads a text line, breaking it into a command name and arguments, and executes the given command. When execution of a command is complete, the Shell resumes execution and displays a "$" prompt to indicate that it is ready for another command.

The following description provides an overview of the Shell facilities that apply to any command that may be invoked by the user. The Shell is a command itself, and is described in detail under the name SH later in this section.

### Command Syntax

A command line consists of a command name (the first word), and its arguments (any other words), which are separated by spaces, as follows

```
command argument1 argument2 ...
```

Multiple commands can be placed on the same line separated by semicolons. Each command is executed in sequence.

The name of any executable program can be used as a command. A new process is created to execute the command, using the code file specified by the command name. If the command name starts with "/", then that exact pathname is used for the code file. Otherwise, the command is searched for first in the current working directory, then in the "/bin" directory (standing for binary programs), and finally in the "/usr/bin" directory where less commonly used commands reside.

The arguments collected by the Shell are made available to the program, which can retrieve them by calling a system interface routine. Certain arguments are interpreted specially by the Shell as described below, and are not passed through to the executing command.

Upon termination each command returns a 32-bit word of status. By convention, the value is zero for successful execution, or nonzero to indicate problems such as invalid or inaccessible arguments or other troubles encountered during execution. This status word is referred to as the "exit code" or "return code", and is described only where special conventions apply. The Shell currently ignores the exit code of all commands.

An example of a simple command is

```
cd /usr/john
```

The Shell will look first for the program "cd" in the current working directory, then for the program "/bin/cd", and then for the program "/usr/bin/cd". Assuming the system-supplied utility command "cd" is found, the current working directory will be changed by this command to the argument "/usr/john".

## File Name Patterns

A command argument that contains the character "*" is treated as a file name pattern by the Shell. Each file name pattern is replaced by all the file names that match the pattern, where each "*" in the pattern matches zero or more characters. The list of matching file names is alphabetically sorted, with each name becoming a separate argument to the command.

For example, if the directory "/exam" contains the following names

```
asm.s
demo
demo.s
model.data
models
test.s
```

then the argument pattern "/exam/*m*s" would be replaced by the following arguments

```
/exam/asm.s /exam/demo.s /exam/models
```

## Standard Input/Output Redirection

Many programs that are run as commands use the terminal for both their input and output. The Shell provides a notation that allows the input or output to be redirected to a file or another device without requiring changes to the program.

Each time a command is run, the Shell uses the aliasing mechanism of the User Monitor to set up a name mapping for the three special files known as standard input, standard output, and standard error output. Standard input has the alias "input", standard output has the alias "output", and standard error output has the alias "stderr". When a file is opened by the program using one of these aliases, the alias is mapped into an actual file name which identifies the target for input or output.

By default, the Shell sets up a mapping for all three aliases to the device name ":termnoecho". Thus the input and output are directed to the terminal unless special arguments interpreted by the Shell are present in the command line. All other files are accessed explicitly by name and generally appear as regular arguments to the command.

The Shell interprets special arguments which set the mapping of standard input and standard output for a command. These special arguments may be used with any command, and are interpreted by the Shell but not passed on as arguments to the command. The symbols "<", ">", "from", or "to" (upper or lower case) followed by a file name indicate a special input/output redirection argument.

An argument of the form "< name" or "from name" indicates that the alias "input" should be mapped into the string "name", thus redirecting standard input.

An argument of the form "> name" or "to name" indicates that the alias "output" should be mapped into the string "name", thus redirecting standard output.

There is no Shell notation for redirecting standard error output ("stderr"), which in general remains mapped to the terminal since it is the target of error messages meant for the interactive user.

For example, the command

```
ls
```

lists the names of the files in the current directory, writing its standard output on the terminal. The command

```
ls >dirlist
```

will create a file called "dirlist" and place the listing there.

## Command Files

A command file is a file that contains lines of commands which can be executed by the Shell. Because the Shell is itself a command (called "sh"), it can be invoked with a command file as its standard input, thus creating a subshell that executes the commands in the file. For example, assume the file "demos" contains the text

```
demo1
demo2
demo3
```

Then the command

```
sh <demos
```

would be the same as if the commands "demo1", "demo2", and "demo3" had been typed at the terminal.

Arguments to the subshell can be referred to in the command file using the positional parameters "$1", "$2", ... , "$n", where "n" is a decimal number. As the command file is interpreted, the parameters are replaced by the corresponding argument strings, where "$1" corresponds to the first argument, "$2" to the second argument, and so on. For example, assume the file "append2" contains the text

```
     cat $1 >temp
     cat temp $2 >$1
     rm temp
```

where "cat" concatenates files together, and "rm" removes files. Then the command

```
     sh <append2 lib subr
```

would be equivalent to

```
     cat lib >temp
     cat temp subr >lib
     rm temp
```

## SUMMARY OF COMMANDS

The following list of system-supplied commands summarizes their usage, and is grouped according to function. Each command is described in detail later in this section, where each command is listed alphabetically in a standard format for easy reference.

### File Manipulation

| | |
|---|---|
| CAT | Concatenate one or more files onto the standard output. Print files on the terminal, or copy them to another device. |
| CMP | Compare two files byte-by-byte and report if they are different. |
| HEXDUMP | Dump arbitrary files in hexadecimal notation, byte-by-byte with the corresponding ASCII characters if printable. |

### Directory Manipulation

| | |
|---|---|
| MKDIR | Make a new directory. |
| RM | Remove files or directories. |
| CD | Change the current working directory. |
| PWD | Print the pathname of the current working directory. |

### Status Inquiry

| | |
|---|---|
| LS | List the names of files or directories within one or more directories. Optional information includes file size, date last modified, permissions, and internal file identifier. |
| DATE | Print or set the current system date and time. |
| VOLMGR.TEST | Query and test the file system. Allows interactive maintenance-level operations. |

### Floppy Disc Maintenance

| | |
|---|---|
| DIR | List floppy disc directory. All files are listed, with file attribute information and volume space usage indicated. |
| ZERO | Clear floppy disc directory. Optionally format disc for use in Ridge system. |
| CRUNCH | Compact floppy disc space allocation. |

Program Development Tools

| | |
|---|---|
| EDIT | Interactive text editor oriented to the Ridge display. TELEDIT is functionally similar, but oriented to the Televideo 950 CRT. Allows creation and modification of program sources, textual data, and other documents. |
| PASC | Compiler for Pascal language source programs. |
| FORT | Compiler for FORTRAN language source programs. |
| PTRANS | Translator for P-code, the intermediate result of the Pascal and FORTRAN compilers. Produces an object module to be linked into an executable program. |
| RASM | Assembler for Ridge assembly language source programs. |
| LINK | Linker that binds several object modules into an executable program. |

Program Execution

| | |
|---|---|
| SH | The Shell, a command interpreter. |

## NOTATION

The following conventions are used in the description of each command's format.

Uppercase words or characters are considered literals that are typed as they appear. Most programs, including the Shell, treat lowercase as equivalent to uppercase, so that it is possible to type literals in either upper or lower case when actually using the commands.

Lowercase words or characters represent arguments that are given a value by the user. Arguments such as "file" or "pathname" refer to a specific file name that will be passed to the command.

Square brackets "[" and "]" surround items which are optional. The brackets are not typed as part of the command; they are merely a descriptive notation.

Ellipses "..." indicate that the preceding item may be repeated one or more times. Ellipses are also only a descriptive notation, and do not appear in the actual command.

Most commands employ a common convention for arguments that indicate flags or other options. Arguments that are preceded by a minus sign "-" are generally interpreted by the command as a flag option rather than a file name. Commands that allow two or more flag options generally accept either each option as a separate argument, or all options concatenated into a single argument. For example:

    -LI would be equivalent to -L -I

CAT - concatenate and copy files

## FORMAT

```
CAT [ file  ... ]
```

## DESCRIPTION

CAT reads each of the input files in sequence and writes them to the standard output. A file can be printed on the terminal (the default for standard output). For example:

```
CAT file1
```

copies file1 to the standard output. Several files can be concatenated together into a single file. For example:

```
CAT file1 file2 > file3
```

concatenates the contents of file1 and file2 together, placing the result in file3.

If no input file is given, the standard input is read and copied to the standard output. For example:

```
CAT > file1
```

copies the standard input (the terminal) to file1. The end of file on the terminal is indicated by typing CONTROL-C.

## NOTES

Beware of using the same file for both input and output, as in "CAT file1 file2 > file1", since opening the output file will truncate it to zero length before it is read.

CD - change current directory

## FORMAT

CD directory

## DESCRIPTION

CD makes directory the new current working directory. This directory becomes the default prefix for any pathname not starting with the root "/".

## SEE ALSO

PWD

CMP - compare two files

## FORMAT

```
CMP [ -L ] [ -S ] file1 file2
```

## DESCRIPTION

CMP compares the two files byte by byte. If no options are given, CMP prints nothing if the files have identical contents. If they differ, the byte number (in decimal) and the values (in hexadecimal) of the first difference are printed on the standard output. If one file is an initial subsequence of the other, that fact is printed.

The -L option causes the byte number and the differing values for each difference (not just the first) to be printed.

The -S option inhibits printing in the case of differing files; only the exit code returned by CMP is of value. Options -S and -L are mutually exclusive (only one may be given).

## NOTES

Exit code 0 is returned for identical files, code 1 for different files, and code 2 for any errors such as missing or invalid arguments or inaccessible files.

CRUNCH - compact floppy disc space allocation

## FORMAT

```
CRUNCH [ -L ] [ -R ]
```

## DESCRIPTION

CRUNCH compacts the space allocation of a floppy disc to maximize usage of the secondary storage. Free space for new files is always allocated at the end of the space in use; unused holes in the space allocation are created when files are deleted or a new version of a file is written. CRUNCH copies files from the end of the disc down over the holes, leaving them in the same order, thus freeing the unused space.

The default disc drive is the left floppy; the -L option specifies the left floppy, and the -R option specifies the right floppy.

## SEE ALSO

DIR, RM, ZERO

## NOTES

An error message (which includes a decimal number returned by the disc driver software) is printed if the disc is not mounted, unreadable or unwriteable, or if the drive is not functioning correctly.

DATE - print or set the date and time

## FORMAT

```
DATE [ yymmddhhmm [ . ss ] ]
```

## DESCRIPTION

DATE prints the current system date and time on the standard output if no argument is given.

If an argument is given, the current date and time are set to that value, and then printed on the standard output. The argument is written as a single number, where "yy" is the last two digits of the year; the first "mm" is the month number; "dd" is the day number in the month; "hh" is the hour number in the 24 hour system; the second "mm" is the minute number; and ".ss" is the seconds, which if omitted defaults to zero. The year, month, and day may be omitted, with the current values being the defaults. For example:

```
DATE 09281748
```

sets the date to Sep 28, 5:48 PM in the current year.

## NOTES

The error message "bad conversion" is printed if the argument given is not in the correct format.

## DIR - list floppy disc directory

### FORMAT

```
DIR [ -L ] [ -R ]
```

### DESCRIPTION

DIR lists the contents of the directory of a floppy disc on the standard output. The listing contains the volume name and date of creation, single/double sides and density information, and the number of blocks on the first line. For each file in the directory, the name, number of blocks, creation date, starting block number, number of bytes in the last block, and the file type are listed on following lines. The last line lists the number of files, plus used and unused blocks in the volume.

The default disc drive is the left floppy; the -L option specifies the left floppy, and the -R option specifies the right floppy.

### SEE ALSO

CRUNCH, RM, ZERO

### NOTES

An error message (which includes a decimal number returned by the disc driver software) is printed if the disc is not mounted, unreadable, or if the drive is not functioning correctly.

## EDIT - display-oriented text editor

### FORMAT

```
EDIT [ file ]
```

### DESCRIPTION

EDIT is the standard text editor for the Ridge display. It uses the special features of the Ridge display and keyboard to support full-screen editing for program text and documents. The set of commands and the function of EDIT are described in the Ridge Text Editor Reference Manual.

If an argument is given, EDIT simulates an "attach" command on that file when the editor is first started. The given file is read in so that it can be edited.

### SEE ALSO

Ridge Text Editor Reference Manual, TELEDIT

## FORT - FORTRAN compiler

### FORMAT

```
FORT [ -L listfile ] [ -O objfile ] file
```

### DESCRIPTION

FORT is the FORTRAN compiler. FORT takes a source input file and compiles it into an intermediate format called P-code. The source file name must end in the extension ".S" (upper or lower case).

The resulting P-code file is suitable as input to the P-code translator PTRANS. The default name of the P-code file is the same as the source name, but with ".P" replacing the ".S". The -O option takes a file name argument which is used to explicitly specify the P-code file name.

The -L option takes a file name argument which is used to specify the name of a file where a listing is written. The listing includes the compiled source, with line numbers, and any syntax errors. No listing is produced by default.

### SEE ALSO

Ridge FORTRAN Reference Manual, LINK, PTRANS

## HEXDUMP - hexadecimal dump

### FORMAT

```
HEXDUMP [ file ... ]
```

### DESCRIPTION

HEXDUMP reads each of the input files in sequence and prints their contents on the standard output in hexadecimal notation. The input bytes are printed 16 to a line, preceded by the byte number within the file. To the right of the hexadecimal values, each byte is also printed in ASCII, with non-printable characters appearing as a dot ".".

If no input file argument is given, the standard input is read and dumped to the standard output.

INVERT - invert display screen

## FORMAT

```
INVERT
```

## DESCRIPTION

INVERT complements the Ridge display screen, changing all black pixels to white, and all white pixels to black. Screen memory bits are not modified; instead, the display hardware is told to change its interpretation of binary 1s and 0s mapping into black and white. This interpretation remains in effect until the next INVERT.

LINK - object module linker

## FORMAT

```
LINK [ -H ] [ -L listfile ] [ -O objfile ] file ...
```

## DESCRIPTION

LINK combines several object files into one, resolving external references between the object modules. The result of LINK is usually an executable code file, with virtual addresses assigned to the code starting at zero.

The input file arguments are concatenated in the order specified, where the first file name must end in the extension ".O" (upper or lower case). The default name of the resulting output file is the same as the first file argument given without the ".O" extension. The -O option takes a file name argument which is used to specify the resulting output file.

The -L option takes a file name argument which is used to specify the name of a file where a listing is written. The listing includes a list of global names with their assigned addresses, sorted both by name and by address. No listing is produced by default.

The -H option specifies that the format of the resulting output file is to remain in "hex" format, which can become the input to a later LINK run. The "hex" format maintains the unresolved external references and global names of the object modules. The default output format is executable binary code.

## SEE ALSO

FORT, PASC, PTRANS, RASM

## LS - list directory or file

### FORMAT

```
LS [ -I ] [ -L ] [ name ... ]
```

### DESCRIPTION

LS lists the contents of the directory for each directory argument. The entries within each directory are listed in alphabetic order. Each file argument is listed with any attributes specified by the options. If no argument is given, the contents of the current working directory are listed. Nothing is listed for arguments which are nonexistent or inaccessible. The listing is written on the standard output.

The -L option indicates long format. The access mode, the number of links, the size in bytes, and the time of last modification are given for each file entry. Only the mode and number of links are given for directory entries.

The access mode of an entry is printed as 10 characters, using the following format. The first character is

| | |
|---|---|
| d | if the entry is a directory, |
| - | if the entry is a file. |

The other 9 characters represent three sets, each with three permission bits. The first set specifies the permissions granted to the owner, the second set specifies the permissions granted to others in the same user group, and the third set specifies the permissions granted to all others. Within each set, from left to right, the three bits grant permission to read, to write, or to execute the file as a program. For a directory, execute permission means the permission to search the directory for a specified file. The permission bits are

| | |
|---|---|
| r | if the file is readable, |
| w | if the file is writeable, |
| x | if the file is executable, |
| - | if the permission is not granted. |

The -I option causes the internal file identifier to be listed in the first column of a file entry.

### NOTES

Currently, the access mode permission bits are ignored by ROS, that is, any file can be used in any way with no checking performed.

MKDIR - make directory

## FORMAT

```
MKDIR directory  ...
```

## DESCRIPTION

MKDIR creates each of the specified directories in the order given.

## SEE ALSO

LS, RM

## PASC - Pascal compiler

### FORMAT

```
PASC [ -L listfile ] [ -O objfile ] file
```

### DESCRIPTION

PASC is the Pascal compiler. PASC takes a source input file and compiles it into an intermediate format called P-code. The source file name must end in the extension ".S" (upper or lower case).

The resulting P-code file is suitable as input to the P-code translator PTRANS. The default name of the P-code file is the same as the source name, but with ".P" replacing the ".S". The -O option takes a file name argument which is used to explicitly specify the P-code file name.

The -L option takes a file name argument which is used to specify the name of a file where a listing is written. The listing includes the compiled source, with line numbers, and any syntax errors. No listing is produced by default.

### SEE ALSO

Ridge Pascal Reference Manual, LINK, PTRANS

PTRANS - P-code translator

## FORMAT

```
PTRANS [ -L listfile ] [ -O objfile ] file
```

## DESCRIPTION

PTRANS takes a P-code input file and translates it into an object file. The P-code file name must end in the extension ".P" (upper or lower case).

The resulting object file is in "hex" format (representing Ridge machine instructions), and is suitable as input to the object module linker LINK. The default name of the object file is the same as the P-code name, but with ".O" replacing the ".P". The -O option takes a file name argument which is used to explicitly specify the object file name.

The -L option takes a file name argument which is used to specify the name of a file where a listing is written. The listing includes an assembly language representation of the translated code. No listing is produced by default.

## SEE ALSO

FORT, LINK, PASC

PWD - print working directory

## FORMAT

PWD

## DESCRIPTION

PWD prints the pathname of the current working directory on the standard output.

## SEE ALSO

CD

RASM - Ridge assembler

## FORMAT

```
RASM [ -L listfile ] [ -O objfile ] file
```

## DESCRIPTION

RASM is the assembler for Ridge assembly language. RASM takes a source input file and assembles it into an object file. The source file name must end in the extension ".S" (upper or lower case).

Depending on an assembler directive placed in the source file, the resulting object file can be created in one of two formats. The object file may be either executable binary code, or it may be in "hex" format which is suitable as input to the object module linker LINK. The default name of the object file is the same as the source name, but with ".O" replacing the ".S". The -O option takes a file name argument which is used to explicitly specify the object file name.

The -L option takes a file name argument which is used to specify the name of a file where a listing is written. The listing includes the assembly language source, with line numbers and the corresponding hexadecimal object code. No listing is produced by default.

## SEE ALSO

Ridge Assembler Reference Manual, LINK

RM - remove files or directories

## FORMAT

```
RM name  ...
```

## DESCRIPTION

RM removes each of the named files or directories. The named entry is deleted from the containing directory and any allocated disc space is made available for subsequent use. A directory must be empty to be removed.

## SEE ALSO

CRUNCH, DIR, LS, MKDIR

SH - shell

## FORMAT

SH [ argument ... ]

## DESCRIPTION

SH is the shell, a command interpreter that executes commands read from the terminal or a file. As part of command interpretation, SH accumulates command arguments, performs parameter substitution, input/output redirection, and file name pattern matching prior to executing each command.

The prompt "$ " is printed on the standard output before each command is read. Commands are read from the standard input, and echoed on the standard output. SH terminates when it encounters end-of-file on its input.

A command is a sequence of nonblank words separated by blanks, where a blank is either a space or a tab character. A semicolon ";" separates commands on the same line, while a return character ends a command and causes each command on the line to be executed sequentially. The first word in each command specifies the file name of the program to be executed. Any remaining words are passed as arguments to the invoked program, unless they are interpreted by the shell as described below. The command name is passed as argument 0.

Within the input, the character "$" is used to indicate a substitutable parameter. A parameter consists of the leading "$" concatenated with a decimal number, which stands for one of the arguments passed to the shell when it was invoked. The correspondence is positional, where "$1" corresponds to the first argument, "$2" to the second argument, and so on. The corresponding argument is substituted for the parameter; if there is no argument, the parameter is replaced by the null string, effectively removing it from the input. Parameter substitution is performed before the input is separated into individual words, thus allowing concatenation of the substitution with nonblank words.

The standard input and standard output of a command may be redirected using a special notation interpreted by the shell. Following parameter substitution, the resulting input is scanned for the special symbols "<", ">", "FROM", or "TO" (upper or lower case), followed by a nonblank word. The symbol "<" or "FROM" indicates that the following word is the name of a file that should become the standard input of the command. The symbol ">" or "TO" indicates that the following word is the name of a file that should become the standard output of the command. The redirection symbol and the file name word are not passed as arguments to the command. Input/output redirection is performed before the command input is separated into individual words.

After parameter substitution, input/output redirection, and separation into individual words, each word is scanned for the character "*", which indicates a file name pattern. Each file name pattern is replaced with all the alphabetically sorted file names that match the pattern, where each "*" in the pattern matches zero or more characters. The character "/" must be matched explicitly as part of a pathname. The special directory names "." and ".." also must be matched explicitly. If no file name is found that matches the pattern, the word is left as is.

A new process is created to execute the command, using the code file specified by the command name. If the command pathname starts with "/", then that exact file name is used. Otherwise, the command is first searched for in the current working directory, then in the "/bin" directory, and finally in the "/usr/bin" directory.

Since SH is a command itself, it is possible to create a subshell to execute commands read from a file. The commands within a shell command file are executed by invoking SH with the command file as its standard input, with any other arguments being used for parameter substitution. For example, if the file "make" contains the text

```
pasc $1.s; ptrans $1.p; link $1.o /lib/rtl.o
```

then the command

```
sh <make prog
```

would cause the following commands to be executed

```
pasc prog.s
ptrans prog.p
link prog.o /lib/rtl.o
```

## NOTES

The message "bad syntax" from SH indicates that an input/output redirection argument is improperly used.

The method of using command files via standard input rather than as an argument is subject to change.

Currently, command files may not be nested more than one level; that is, the SH command should not appear within a command file.

TELEDIT - TeleVideo-oriented text editor

## FORMAT

```
TELEDIT [ file ]
```

## DESCRIPTION

TELEDIT is a text editor for the Televideo 950 CRT. It is similar to EDIT, which is oriented to the Ridge display, but uses different function keys and other specific features of the Televideo, allowing editing from an RS-232 device. The set of commands and the function of TELEDIT are described in the Ridge Text Editor Reference Manual.

If an argument is given, TELEDIT simulates an "attach" command on that file when the editor is first started. The given file is read in so that it can be edited.

## SEE ALSO

Ridge Text Editor Reference Manual, EDIT

VOLMGR.TEST - test and query file system

## FORMAT

VOLMGR.TEST

## DESCRIPTION

VOLMGR.TEST is used to test file system integrity, list file allocation and disc usage, and perform maintenance-level operations on the file system. VOLMGR.TEST is an interactive program, using the standard input and output, with a set of self-explanatory commands and prompts.

## NOTES

Queries and other requests (including updating file attributes) are performed by directly interacting with the Volume Manager, and thus should be used with caution to avoid destroying the file system structure on disc.

## ZERO - clear floppy disc directory

### FORMAT

```
ZERO [ -F ] [ -L ] [ -R ] volumename
```

### DESCRIPTION

ZERO clears the directory of a floppy disc; that is, all file entries are removed from the volume, leaving an empty directory. The volume is given the indicated name (up to 7 characters).

The -F option causes the floppy disc to be formatted first; a floppy disc must be formatted before other usage in a Ridge system.

The default disc drive is the left floppy; the -L option specifies the left floppy, and the -R option specifies the right floppy.

### SEE ALSO

CRUNCH, DIR

### NOTES

An error message (which includes a decimal number returned by the disc driver software) is printed if the disc is not mounted, unreadable or unwriteable, or if the drive is not functioning correctly.

# SECTION 3

# KERNEL INTERFACE

The Kernel is the lowest level of the Ridge Operating System. The Kernel implements process creation and deletion, message-oriented interprocess communication, memory management, and multitasking primitives. In addition, it provides the first level handling of program traps and faults, and dispatches input/output device drivers when an interrupt occurs.

## KERNEL ORGANIZATION

The Kernel routines are the only routines in the system that run in a special processor mode called "kernel mode". Kernel mode is used to perform all privileged activity that involves data sharing or synchronization. In kernel mode, the processor uses real memory addresses instead of performing address translation of virtual addresses. Certain privileged processor instructions are valid only in kernel mode, thus allowing the Kernel to protect user and system processes from interfering with each other.

All other routines in the system run in "user mode" which uses virtual addresses. User and system processes request Kernel services via a special KCALL instruction. The KCALL instruction causes the processor to switch from user mode to kernel mode and enter the appropriate Kernel routine based on the instruction operand. The calling process is suspended for the duration of the KCALL instruction. When the Kernel routine is completed, the processor returns to user mode and execution proceeds.

The Kernel is also entered whenever an "exception" occurs. An exception is either a user mode program fault or trap, or an external interrupt. Exception handling is described later in this section.

### Process Management

The Kernel provides routines to create, delete and manage "processes". A process is an instance of a running program that performs some computational task or function. The Ridge Operating System is composed of several system processes that manage various system resources and provide a service interface to other client processes. Each user program is run as an individual process that interacts with the system processes to accomplish its task.

A process runs in its own private environment, which includes a code segment, a data segment and a queue segment, plus process state information maintained in a process control block managed by the Kernel. The code segment contains the executable processor instructions for the process, while the data segment contains the data (variables, stack, heap, etc.). The queue segment is used for interprocess communication and is only addressable by the Kernel.

Each code, data or queue segment is a full 32-bit addressable virtual memory address space, which gives a maximum size of four gigabytes that is demand-paged in 4096-byte blocks. The secondary storage for a segment is usually a file, which is specified using a file identifier known to the file system. The executable code file for a process serves as the code segment, while the data and queue segments are usually temporary files that exist only for the life of the process.

A process is created via the CreateProcess routine. The code, data, and queue segments are specified, as well as the initial program counter and traps word. A process priority is specified which is used by the Kernel when scheduling which process is to be run. Smaller numbers indicate higher priority. A process ID (PID) is returned by CreateProcess, which is used in further requests to manipulate the new process. The process which called CreateProcess becomes the "parent" of the new process, and may receive special messages related to the status of the "child" process.

The routines Activate and Suspend can be used to make a process active or inactive. A process is deleted by the Kill routine, or it may delete itself by calling the Terminate routine.

The state of each process is maintained in a memory-resident set of process control blocks which is managed by the Kernel. The process state includes the contents of the general registers, the program counter, the traps word, the amount of processor time used, and other status information. The routines ReadProcessState and WriteProcessState allow access to the state of a process.

A process can determine its own PID via the MyID routine. The PID of special system processes can be found via the GetSpecialPID routine, while SetSpecialPID is used by special processes to register themselves.

The Kernel maintains a "ready list" of the processes that are eligible to run, sorted by process priority. All other processes are blocked from running for various reasons, such as encountering a fault, or waiting for a message. When the conditions causing a process to be blocked are removed, the process is inserted into the ready list after any other processes of the same priority.

The Kernel simply runs the highest priority process. Process scheduling is handled by adjusting the priorities of processes or by causing the ready list to be modified by activating or suspending processes.

## Process Communication

Processes communicate with each other via "messages". Messages are fixed-size blocks of data that are transferred from one process to another via Kernel routines. The message primitives are optimized to make fast transfer of information between processes possible.

Processes use messages to synchronize their activities as well as to share data with each other. The message mechanisms are also used by the Kernel to indicate events such as external interrupts, faults, or traps encountered by a process.

The queue segment of each process contains the data structures which support messages. The data structures include "links" and "queues", plus message buffers, which are accessed only by the Kernel routines. A link and queue form a unidirectional channel between two processes. The link is an outbound channel from one process to a queue of another process, while the queue is an inbound channel that may receive messages via one or more links.

Messages are buffered in a first-in-first-out manner, using data structures and buffers maintained in the queue segment. Since the message resources are allocated on a per process basis, the number of links, queues, and message buffers can be matched to the needs of a particular process. The InitQSeg routine is used to initialize a queue segment by specifying the number of links, queues, and message buffers for a process.

In order to receive messages, a process must create a queue via the OpenQueue routine. In order to send messages, a process must create a link to the specific receiver's queue via the OpenLink routine. Messages may only be sent over valid links to a specific process, so a receiving process does not have to be concerned with illegitimate messages or message buffer overflow generated by an unknown sender. Queues can be deleted via the CloseQueue routine, while links may be deleted via CloseLink.

Messages can be in one of two forms. Short 32-byte messages are exchanged using the Send and Receive routines. Larger data transfer is accomplished by using SendPage and ReceivePage, which include a 4096-byte page in addition to a 32-byte message. The Test routine allows a process to check if a queue has any messages.

When a message is sent, the Kernel accesses the queue segment of the sending process to find the specified link data structure. The link indicates where the queue segment of the receiving process is located. The Kernel then acts as a bridge between the two separate address spaces by transferring the message into the receiver's queue. The message is retrieved from the queue segment of the receiver when the process calls the Receive or ReceivePage routine.

Context switching from process to process is handled by the Kernel as a side effect of process communication primitives. Attempting to receive a message on an empty queue causes the receiver to be suspended until a message appears, allowing other processes to run. Sending a message does not suspend the sender who is free to resume execution immediately after the Send or SendPage routine returns. However, if the sender's priority is lower than a receiver who is waiting for a message, then the Kernel performs a context switch to the receiving process. External interrupts may also cause context switching. Interrupt handling is described later in this section.

A process may need to wait for messages in several queues. Each queue can be armed to provide a process wakeup trigger via the Arm routine, and then the process can call the Wait routine. The process will be suspended until a message appears at one of the armed queues, and then it will be activated once all higher priority processes become suspended. Each queue can then be tested, and any outstanding messages can be received. A queue can be disarmed via the Disarm routine.

## Memory Management

The Kernel provides memory management routines that assist the system in implementing virtual memory. The Kernel manages the Virtual to Real Translation (VRT) table that provides a mapping of virtual addresses to real memory addresses. Most of the memory management routines are privileged routines that are only meant to be called by the Virtual Memory Manager. These routines are therefore not described here.

The Kernel is the first level handler for page faults, which are references to a virtual memory page that is not currently mapped to a real memory page. When a process encounters a page fault, it is suspended and the Kernel causes a message to be sent to the Virtual Memory Manager. After the fault has been handled by causing the appropriate page to be read in from secondary storage, the Virtual Memory Manager calls a Kernel routine that makes the faulting process ready to resume execution.

In some cases while executing a Kernel routine entered via a KCALL instruction, the Kernel may need to reference a virtual memory address of a process and thus may cause a page fault. When this happens, the registers are restored and the user process program counter is reset to point to the KCALL instruction. Thus, when the page fault is handled and the process is resumed, the KCALL will be executed again. By restarting the KCALL from the beginning the process state manipulation is greatly simplified, and all Kernel operations can be treated as atomic.

There are several memory management routines that may be applicable to special processes such as device drivers. The Trans routine is used to translate a virtual address into a real address. A real memory address might be necessary when setting up a peripheral device. The Flush routine causes a portion of the address space of a process to be written to secondary storage. The pages of memory that have become "dirty" due to write references can be flushed to secondary storage, for instance, to insure that changes to data structures also appear on a non-volatile storage medium.

The Fix routine is used to lock a virtual page in memory, making it always resident. Locking part of the queue segment is necessary to insure that messages generated by the Kernel are not lost for special events such as external interrupts or faults. Pages can be unlocked by the Free routine.

## Exception Handling

Abnormal execution of a processor instruction is termed an "exception". This may be caused by an error in the instruction which causes a trap, by a page fault, by an external interrupt from a peripheral device, or by some other unusual condition. The occurence of an exception will cause the processor operating in user mode to switch to kernel mode and begin execution of a specific Kernel routine which depends on the exception. See the Ridge Processor Reference Manual for a description of the possible exceptions and the processor's response.

The Kernel provides the first level handling of interrupts from peripheral devices by getting the I/O Interrupt Read word from the hardware which includes the device ID. A device driver process can associate itself with a particular device via the AcquireDevice routine and then automatically receive a message from the Kernel when an interrupt occurs. The driver's queue must be locked in memory so that messages are not discarded.

When an interrupt occurs for a device that has an associated interrupt process, the Kernel formats a message containing the device ID and the I/O Interrupt Read word and places it into the queue of the interrupt process. If the interrupt occurs for a process of higher priority than the currently executing process, the current process is suspended and the interrupt process begins executing. The ReleaseDevice routine breaks the association of an interrupt process with a device.

An error encountered during the execution of an instruction generally causes a trap. Some traps can be enabled or disabled by the traps word which is part of the state of a process. The traps word of a process is initialized as part of the CreateProcess routine, and can be accessed by the ReadProcessState and WriteProcessState routines.

When a trap occurs, the current process is suspended and then the Kernel sends a message to the parent process of the process that caused the trap. The parent process is responsible for handling the trap, and may resume the child process via Activate, or decide to Kill the process depending on the trap.

All other exceptions, such as page faults, power fail warnings, Switch 0 interrupts, and timer interrupts are handled by the Kernel in special ways and are not described here.

## Time Functions

The Kernel provides several routines that allow access to time functions supported by the processor. The Runtime routine returns the current number of milliseconds used by the calling process. The TimeOfDay routine returns a 64-bit number that represents the system date and time in terms of nanoseconds since the start of the year 1900. The 64-bit "timestamp" can be used to create unique internal identifiers. The SetTimeOfDay routine allows the system date and time to be set.

## KERNEL INTERFACE DATA STRUCTURES

The Kernel interface routines have arguments that can be defined in terms of several data structures. The following Pascal type definitions describe these data structures, and are assumed in the description of each of the routines found later in this section.

The packing option must be used in Pascal programs that call Kernel routines which use types that are affected by the packing option, such as Boolean or Halfword. See the Ridge Pascal Reference Manual for information on packing of data.

```
     Halfword        = 0..65535;
```

A Halfword is 16 bits.

```
     Double          = Set of 0..63;
```

A Double is a 64-bit double word that is normally treated as just a string of bits.

```
     Error           = Integer;
```

An Error is a status that is returned by most interface routines. The value zero indicates successful completion, while nonzero values indicate some failure. Each routine may define specific values to indicate certain errors.

```
     ProcessID       = Integer;
```

A ProcessID is an identifier for a process which is returned when a process is created. Further manipulation of the process requires the ProcessID.

```
     Link            = Integer;
```

A Link is an identifier for an outbound channel which is used to send a message to another process.

```
     Queue           = Integer;
```

A Queue is an identifier for an inbound channel which is used to receive messages from other processes.

```
     Message         = Array [0..7] of Integer;
```

A Message is 8 words of data that are sent or received by the communication routines. The Message array must be double-word aligned (start at an address that is a multiple of 8).

```
     VirtualAddress = Integer;
```

A VirtualAddress is a virtual address within a code, data or queue segment.

```
     PageAddress     = Integer;
```

A PageAddress contains the virtual address of a 4096-byte page, which must be aligned on a page boundary.

```
     SegmentID       = Integer;
```

A SegmentID identifies either the code, data or queue segment of a process. The value 0 specifies the code segment, 1 specifies the data segment, and 2 specifies the queue segment.

```
     DeviceNumber    = Integer;
```

A DeviceNumber identifies a peripheral device. The possible values 0 to 255 correspond to the device numbers that are set on a device controller and are used by the input/output hardware.

```
     TimeStamp       = Double;
```

A TimeStamp is an encoding of a date and time. It represents the number of nanoseconds since the beginning of the year 1900.

```
     FileID          = Double;
```

A FileID is an internal file identifier used by the file system to specify a particular file.

```
     Traps           = Set of
                         (TDebug,T1,T2,T3,T4,T5,T6,T7,T8,
                         T9,T10,T11,T12,T13,T14,T15,
                         TIntOvflow,TIntDiv0,TRealOvflow,
                         TRealUflow,TRealDiv0,T21,T22,
                         T23,T24,T25,T26,T27,T28,T29,T30,
                         TPriv);
```

The Traps word contains 32 bits that enable or disable some processor exceptions for a each process (refer to the Ridge Processor Reference Manual). If the named bit is in the set (represented by a 1 bit), then the corresponding trap is enabled.

The TPriv bit is enabled to allow a process to execute privileged instructions or call privileged Kernel routines. The TIntOvflow, TIntDiv0, TRealOvflow, TRealUflow, and TRealDiv0 bits enable traps on integer and floating point arithmetic operations resulting in overflow, underflow, or division by zero. The TDebug bit is used with the TRAP processor instruction by a debugger to implement software breakpoints.

The other bits are currently unused by ROS.

```
     PProcessState   = ^ ProcessState;
     ProcessState    = Record
                         register : Array [0..15] of Integer;
                         pc       : Integer;
                         parent   : ProcessID;
                         trapBits : Traps;
                         runTime  : Integer;
                         status   : Integer;
                         priority : Integer;
                         msgWait  : Integer;
                       end;
```

The ProcessState structure is used to access information about the state of a process. The structure must be double-word aligned, and must not cross a page boundary.

The "register" array contains the contents of the general registers, while "pc" contains the current program counter. The "parent" is the process which created the specified process, and may be notified in the event of a fault or termination of the child process. The "trapBits" contains the traps word, and "runTime" contains the number of milliseconds of processor time used by the process.

The "status" of a process is only meaningful to the Kernel and the virtual memory system. The "priority" of a process is used when scheduling processes, with smaller numbers indicating higher priority. The "msgWait" flag is nonzero if the process is waiting for a message.

## KERNEL ROUTINES

The following list of Kernel interface routines summarizes their usage, and is grouped according to function. Each routine is described in detail later in this section, where each routine is listed alphabetically in a standard format for easy reference.

Certain routines may only be executed by "privileged" processes, and are so indicated in the detailed descriptions. Privileged processes must have the privileged mode bit "TPriv" on in the traps word. Attempted execution of a privileged routine by a non-privileged process results in an error being returned from the routine to the caller.

### Process Management

| | |
|---|---|
| CreateProcess | Create a new process. Specify a code, data, and queue segment, plus initial program counter, traps word, and priority. |
| Activate | Activate a process, allowing it to resume execution. |
| Suspend | Suspend a process, making it ineligible to execute. |
| Kill | Delete a process. |
| Terminate | Delete the process of the caller, returning an exit code to the parent process. |
| ReadProcessState | Get the current state information of a process. |
| WriteProcessState | Modify the current state information of a process. |
| MyID | Get the process ID of the caller. |
| GetSpecialPID | Get the process ID of a special process. |
| SetSpecialPID | Register the process ID of a special process with the Kernel. |

### Process Communication

| | |
|---|---|
| InitQSeg | Initialize the queue segment of a process. Specify the number of links, queues, messages and pages to allocate. |
| OpenLink | Open a message link to a process. |

| | |
|---|---|
| CloseLink | Close a link. |
| OpenQueue | Open a queue for receipt of incoming messages. |
| CloseQueue | Close a queue. |
| Send | Send a message to a process. |
| SendPage | Send a message and a data page to a process. |
| Receive | Receive a message from a process. |
| ReceivePage | Receive a message and a data page from a process. |
| Test | Check if a message is available in a queue. |
| Arm | Arm a queue to trigger a wakeup when a message arrives on the queue. |
| Disarm | Disarm the wakeup trigger from a queue. |
| Wait | Wait for a message to arrive. Suspends the process until some armed queue receives a message. |
| DeleteMessage | Delete a message from a queue. |

Memory Management

| | |
|---|---|
| Fix | Fix a virtual page in memory. Locks a page within the code, data, or queue segment into real memory. |
| Free | Free a locked page in memory. |
| Flush | Flush a part of virtual memory to secondary storage. Forces any modified pages to be written out. |
| Trans | Translate a virtual address to a real memory address. |

Device Management

| | |
|---|---|
| AcquireDevice | Acquire an external device, associating it with the calling process. Specify a queue for receiving interrupt messages. |
| ReleaseDevice | Release an external device. |

## Time Functions

| | |
|---|---|
| Runtime | Get the process runtime in milliseconds. |
| TimeOfDay | Get the current system date and time. |
| SetTimeOfDay | Set the current system date and time. |

## Assembler Interface Notation

The definition of the Kernel interface routines is given in Pascal. In addition, each description contains an assembly language level interface specification.

Kernel functions are invoked by the KCALL instruction, which has an operand with a value from 0 to 255 which specifies which operation is to be performed. Parameters are passed via the general registers R0 through R11, which may be modified by the Kernel. Registers R12 through R15 remain unmodified by any KCALL. The Pascal-callable routines are actually small assembly language routines that copy the parameters from the Pascal stack to the registers, execute the appropriate KCALL, and then return to the caller.

In the ASSEMBLER part of each routine's description, the KCALL operand number is given in decimal, preceded by the input parameters and followed by the output parameters. The parameters are given with the name of the register followed by the associated parameter name from the Pascal definition. Register R11 always contains the status after a KCALL, which corresponds to the "Error" value returned by a Pascal function.

## Error Return Codes

In the detailed description of each routine, error return codes are specified symbolically. An error code always starts with the characters "Er", as in "ErQueueFull". The mapping of these symbols to actual integer values, plus a brief description of each error, is contained in the Appendix.

## AcquireDevice - acquire an external device

### FORMAT

```
FUNCTION AcquireDevice (dev : DeviceNumber;
                        interruptQ : Queue) : Error;
```

### DESCRIPTION

AcquireDevice is used by a process to notify the Kernel that it wants to receive interrupt messages from a particular peripheral device. The device number "dev" is a number from 0 to 255 that corresponds to the hardware device identifier of a peripheral controller.

A queue "interruptQ" is specified where interrupt messages should be placed. To guarantee that interrupt messages are not discarded, the portion of the queue segment that contains the interrupt queue data structures (usually page zero at least) must be locked in memory.

### ASSEMBLER

```
R0: dev
R1: interruptQ
KCALL 41
```

### SEE ALSO

Fix, ReleaseDevice

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvDevice is returned if the device number is not valid, while ErDeviceInUse is returned if the device has already been acquired by another process.

ErInvQueue is returned if the interrupt queue is not valid.

## Activate - activate a process

### FORMAT

```
FUNCTION Activate (pID : ProcessID) : Error;
```

### DESCRIPTION

Activate will make a suspended process "pID" eligible to resume execution. A process is inactive when first created, and must be activated before it begins running. A process may become inactive as the result of a Suspend call.

A newly activated process is inserted on the ready list after all other processes of the same priority. If the process to be activated is higher priority than the caller, the newly activated process will start executing instead of the caller.

A process which is waiting for a page fault to complete or a message to appear will not start running until the condition it is waiting for becomes satisfied. It is not considered an error to Activate an already active process.

### ASSEMBLER

```
R0: pID
KCALL 16
```

### SEE ALSO

CreateProcess, Kill, Suspend

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

## Arm - arm a queue to trigger wakeup

### FORMAT

```
FUNCTION Arm (q : Queue) : Error;
```

### DESCRIPTION

A process may need to wait for a message to appear in one of several different queues at the same time. Arm is called for each queue "q" that is to trigger a wakeup. Then the process calls the Wait routine, and will be suspended until a message appears in one of the armed queues.

In order for a queue to retain its armed status, the Test routine must be used after waiting to insure that a message exists before the Receive or ReceivePage routine is called.

### ASSEMBLER

```
R8: q
KCALL 6
```

### SEE ALSO

Disarm, Receive, Test, Wait

### NOTES

ErInvQueue is returned if the queue is not valid.

## CloseLink - close a link

### FORMAT

```
FUNCTION CloseLink (pID : ProcessID;
                    l : Link) : Error;
```

### DESCRIPTION

CloseLink closes the link "l" for the process "pID".

Any messages sent on the link prior to the CloseLink call may still be received by the receiving process. A link is automatically closed if the corresponding queue is closed.

### ASSEMBLER

```
R0: pID
R1: l
KCALL 22
```

### SEE ALSO

CloseQueue, OpenLink

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErInvLink is returned if the link is not valid.

## CloseQueue - close a queue

### FORMAT

```
FUNCTION CloseQueue (pID : ProcessID;
                     q : Queue) : Error;
```

### DESCRIPTION

CloseQueue closes the queue "q" for the process "pID".

Any messages in the queue waiting to be received will be discarded. All links to the queue are closed automatically, which involves modifying the queue segment of each process linked to the queue.

### ASSEMBLER

```
R0: pID
R1: q
KCALL 21
```

### SEE ALSO

CloseLink, OpenQueue

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErInvQueue is returned if the queue is not valid.

## CreateProcess - create a new process

### FORMAT

```
FUNCTION CreateProcess (codeFile : FileID;
                        dataFile : FileID;
                        queueFile : FileID;
                        pc : Integer;
                        priority : Integer;
                        trapBits : Traps;
                    var pID : ProcessID) : Error;
```

### DESCRIPTION

CreateProcess creates a new process. The code, data, and queue segments for the process are specified as the file identifiers "codeFile", "dataFile", and "queueFile", respectively.

The process is initially suspended, and the contents of the general registers are undefined. The initial program counter is specified as "pc", which is the address in the code segment where execution begins when the process is activated. The initial traps word is given by "trapBits". Only a privileged process can create another privileged process.

An initial process priority is specified in "priority" which is used when scheduling which process is to be run. Smaller values indicate higher priority.

The process identifier "pID" is returned, which is used in further requests to manipulate the process. The process which calls CreateProcess becomes the "parent" of the new process, and may receive special messages related to the status of the "child" process.

### ASSEMBLER

```
R0,R1: codeFile
R2,R3: dataFile
R4,R5: queueFile
R6: pc
R7: priority
R8: trapBits
KCALL 15
R0: pID
```

### SEE ALSO

Activate, InitQSeg, Kill, Suspend

NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErNoProcBlock is returned if all of the process control blocks are in use.

ErNoFreeSegments is returned if all of the segment identifiers are in use.

## DeleteMessage - delete a message from a queue

### FORMAT

```
FUNCTION DeleteMessage (q : Queue) : Error;
```

### DESCRIPTION

DeleteMessage deletes a message from queue "q". The next message to be received, along with any associated page of data, is discarded.

Typically, DeleteMessage is used when a message arrives with a data page attached when no page is expected, and the receiver does not want to receive the page.

### ASSEMBLER

```
R8: q
KCALL 35
```

### SEE ALSO

Receive, Test

### NOTES

ErInvQueue is returned if the queue is not valid.

ErQueueEmpty is returned if the queue is empty.

## Disarm - disarm the trigger from a queue

### FORMAT

```
FUNCTION Disarm (q : Queue) : Error;
```

### DESCRIPTION

Disarm removes the wakeup trigger for queue "q". After Disarm returns, newly arriving messages on the specified queue will not cause the process which calls Wait to wakeup.

### ASSEMBLER

```
R8: q
KCALL 7
```

### SEE ALSO

Arm, Wait

### NOTES

ErInvQueue is returned if the queue is not valid.

## Fix - fix a page in memory

### FORMAT

```
FUNCTION Fix (seg : SegmentID;
              addr : VirtualAddress) : Error;
```

### DESCRIPTION

Fix locks the virtual page which contains the address "addr" into real memory. The specified page is in the code segment ("seg" is 0), the data segment ("seg" is 1), or the queue segment ("seg" is 2) of the calling process.

The locked virtual page remains in memory, and does not move or become unmapped by the virtual memory system.

### ASSEMBLER

```
R2: seg
R3: addr
KCALL 37
```

### SEE ALSO

AcquireDevice, Free, Trans

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvSeg is returned if the segment selector is other than 0, 1, or 2.

ErNoFreePages is returned if not enough real memory pages would remain as a result of fixing the desired page.

Flush - flush virtual memory to secondary storage

FORMAT

```
FUNCTION Flush (seg : SegmentID;
                lowAddr : VirtualAddress;
                highAddr : VirtualAddress) : Error;
```

DESCRIPTION

Flush causes a portion of the virtual memory segment "seg" to be written to its corresponding secondary storage. The data segment (if "seg" is 1) or the queue segment (if "seg" is 2) of the calling process is flushed. The code segment is never modified, so Flush does nothing if "seg" is 0.

Any pages that include the addresses from "lowAddr" to "highAddr", inclusively, are written out if any modification to them has occurred in real memory. This forces the secondary storage to reflect the updated contents of those pages.

ASSEMBLER

```
R2: seg
R3: lowAddr
R4: highAddr
KCALL 34
```

NOTES

ErInvSeg is returned if the segment selector is other than 0, 1, or 2.

Free - free a page in memory

FORMAT

```
FUNCTION Free (seg : SegmentID;
               addr : VirtualAddress) : Error;
```

DESCRIPTION

Free unlocks the virtual page which contains the address "addr" from real memory. The specified page is in the code segment ("seg" is 0), the data segment ("seg" is 1), or the queue segment ("seg" is 2) of the calling process.

The unlocked virtual page becomes available for use by the virtual memory system.

ASSEMBLER

```
R2: seg
R3: addr
KCALL 38
```

SEE ALSO

Fix

NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvSeg is returned if the segment selector is other than 0, 1, or 2.

ErPageNotReserved is returned if the page was not locked.

GetSpecialPID - get a special process ID

FORMAT

```
FUNCTION GetSpecialPID (genericID : Integer;
                    var pID : ProcessID) : Error;
```

DESCRIPTION

GetSpecialPID allows a process to find the process ID of certain special system processes. The "pID" of the Volume Manager is returned if "genericID" is 1, while the "pID" of the Directory Manager is returned if "genericID" is 2.

The allocation of the generic identifiers must be coordinated by the processes using them as the Kernel attaches no meaning to them. This interface is subject to change.

ASSEMBLER

```
R0: genericID
KCALL 44
R0: pID
```

SEE ALSO

MyID, SetSpecialPID

NOTES

ErBadSpecialPID is returned if the generic ID is invalid.

## InitQSeg - initialize a queue segment

### FORMAT

```
FUNCTION InitQSeg (pID : ProcessID;
                   numLinks : Integer;
                   numQueues : Integer;
                   numMsgs : Integer;
                   numPages : Integer) : Error;
```

### DESCRIPTION

InitQSeg initializes the queue segment for the process "pID". The queue segment data structures must be initialized before any interprocess communication with the process can take place, and before any queues or links are opened.

The maximum number of outgoing links which may be opened for the process is specified in "numLinks". The maximum number of queues which may be opened for the process is specified in "numQueues".

The maximum number of messages which can be buffered by all of the queues open at one time is specified in "numMsgs". Each open queue is allocated a given number of these message buffers by OpenQueue.

The maximum number of data pages associated with messages which can be buffered at one time is specified in "numPages". The pool of data page buffers is shared dynamically by all the open queues of a process.

### ASSEMBLER

```
R0: pID
R1: numLinks
R2: numQueues
R3: numMsgs
R4: numPages
KCALL 24
```

### SEE ALSO

CreateProcess, OpenLink, OpenQueue

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErTooManyLinks is returned if the given number of links will not fit in a page. Currently, the maximum number of links is 255.

ErTooManyQueues is returned if the given number of queues will not fit in a page. Currently, the maximum number of queues is 127.

ErTooManyPages is returned if the page pool data structure will not fit in a page. Currently, the maximum number of pages is 255.

## Kill - delete a process

### FORMAT

```
FUNCTION Kill (pID : ProcessID) : Error;
```

### DESCRIPTION

Kill destroys the specified process "pID". The process may be active or suspended, and is terminated immediately. All links and queues owned by the process are closed.

A message is sent to the parent of the process indicating the process has terminated.

### ASSEMBLER

```
R0: pID
KCALL 18
```

### SEE ALSO

CreateProcess, Terminate

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

MyID - get the process ID of the caller

## FORMAT

```
FUNCTION MyID : ProcessID;
```

## DESCRIPTION

MyID returns the process ID of the caller.

## ASSEMBLER

```
KCALL 23
R0: MyID
```

## SEE ALSO

GetSpecialPID, SetSpecialPID

## OpenLink - open a link

### FORMAT

```
FUNCTION OpenLink (sender : ProcessID;
                   receiver : ProcessID;
                   q : Queue;
               var l : Link) : Error;
```

### DESCRIPTION

OpenLink creates a link from the sending process "sender" to the queue "q" in the receiving process "receiver". Messages may then be sent on the link "l" returned by OpenLink.

### ASSEMBLER

```
R0: sender
R1: receiver
R2: q
KCALL 20
R0: l
```

### SEE ALSO

CloseLink, InitQSeg, Send

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if either the sender or receiver process IDs are invalid.

ErInvQueue is returned if the receiver's queue is not valid.

ErNoFreeLinks is returned if there is no unused link in the sender.

## OpenQueue - open a queue

### FORMAT

```
FUNCTION OpenQueue (pID : ProcessID;
                    numMsgs : Integer;
                var q : Queue) : Error;
```

### DESCRIPTION

OpenQueue creates a queue in the process "pID" which is used for receiving messages. The queue holds messages sent over a link which have not yet been received. The number of message buffers to allocate for the queue is specified in "numMsgs".

The queue "q" is returned, which is used to receive messages from processes that have links established to the open queue. The queue may have any number of links attached to it.

### ASSEMBLER

```
R0: pID
R1: numMsgs
KCALL 19
R0: q
```

### SEE ALSO

CloseQueue, InitQSeg, Receive

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErNoFreeQueues is returned if there is no unused queue in the receiver.

ErTooManyMessages is returned if there are not enough unused message buffers.

ReadProcessState - get the state of a process

## FORMAT

```
FUNCTION ReadProcessState (pID : ProcessID;
                           state : PProcessState) : Error;
```

## DESCRIPTION

ReadProcessState is used to determine the current state of process "pID". The process state includes the general registers, program counter, and the traps word of the process. The parent process, the priority, process runtime, and other status is also returned.

ReadProcessState places a block of information into the data area specified by "state". The ProcessState record is described earlier in this section.

## ASSEMBLER

```
R0: pID
R1: state
KCALL 39
```

## SEE ALSO

WriteProcessState

## NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErBadPCBPointer is returned if the ProcessState record is improperly aligned, or the data structure crosses a page boundary.

## Receive - receive a message

### FORMAT

```
FUNCTION Receive (q : Queue;
              var msg : Message;
              var sender : ProcessID) : Error;
```

### DESCRIPTION

Receive returns the next message "msg" from the queue "q". The process ID of the sending process is returned in "sender".

If the message had no data page associated with it, the message is then removed from the queue. If the message did have a data page associated with it, the message portion is returned to the caller but an error is given. The message is not removed from the queue so that it can be returned by a subsequent ReceivePage call.

If the queue has no messages in it, then the calling process is suspended until a message arrives.

### ASSEMBLER

```
R8: q
KCALL 2
R0-R7: msg
R8: sender
```

### SEE ALSO

ReceivePage, Send, SendPage, Test

### NOTES

Attempting to receive a message on an empty queue that is armed for wakeup will lose the armed trigger status.

ErInvQueue is returned if the queue is not valid.

ErPageSent is returned if the message returned has a data page associated with it.

## ReceivePage - receive a message and a data page

### FORMAT

```
FUNCTION ReceivePage (q : Queue;
                  var msg : Message;
                      dataPage : PageAddress;
                  var sender : ProcessID) : Error;
```

### DESCRIPTION

ReceivePage returns the next message "msg" from the queue "q", plus any associated data page. The process ID of the sending process is returned in "sender".

If the message had a data page associated with it, the contents of the page is placed at the address specified in "dataPage". If the message had no data page associated with it, the message portion is returned to the caller and an error is given. In either case, the message is then removed from the queue.

If the queue has no messages in it, then the calling process is suspended until a message arrives.

### ASSEMBLER

```
R8: q
R9: dataPage
KCALL 4
R0-R7: msg
R8: sender
```

### SEE ALSO

Receive, Send, SendPage, Test

### NOTES

Attempting to receive a message on an empty queue that is armed for wakeup will lose the armed trigger status.

ErInvQueue is returned if the queue is not valid.

ErNoPageSent is returned if the next message in the queue does not have a data page associated with it.

ErNotPageBound is returned if the page address is not aligned on a page boundary.

ReleaseDevice - release an external device

## FORMAT

```
FUNCTION ReleaseDevice (dev : DeviceNumber) : Error;
```

## DESCRIPTION

ReleaseDevice breaks the association of the calling interrupt handling process with the peripheral device "dev". The device number is a value from 0 to 255 that corresponds to the hardware device identifier of a peripheral controller.

After ReleaseDevice returns, no more interrupt messages for the device will be sent to the calling process. The calling process can then unlock pages associated with interrupt messages or input/output for the device.

## ASSEMBLER

```
R0: dev
KCALL 42
```

## SEE ALSO

AcquireDevice, Free

## NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvDevice is returned if the device number is invalid or not acquired by the calling process.

Runtime - get the process runtime

## FORMAT

```
FUNCTION Runtime : Integer;
```

## DESCRIPTION

Runtime returns the runtime of the calling process. The runtime is the number of milliseconds used by the process since its creation.

## ASSEMBLER

```
KCALL 25
R0: Runtime
```

## SEE ALSO

TimeOfDay

Send - send a message

## FORMAT

```
FUNCTION Send (l : Link;
           var msg : Message) : Error;
```

## DESCRIPTION

Send sends the message "msg" using the link "l". The message is placed in the queue of the receiving process.

If the receiving process is waiting for a message on the queue, and is higher priority than the sending process, then the sender is suspended.

## ASSEMBLER

```
R0-R7: msg
R8: l
KCALL 1
```

## SEE ALSO

Receive, ReceivePage, SendPage

## NOTES

ErInvLink is returned if the link is not valid.

ErRcvrDead is returned if the receiving process has terminated.

ErQueueFull is returned if the receiver's queue is full, and the message is discarded.

SendPage - send a message and a data page

## FORMAT

```
FUNCTION SendPage (l : Link;
               var msg : Message;
                   dataPage : PageAddress) : Error;
```

## DESCRIPTION

SendPage sends the message "msg" and the associated data page specified by "dataPage" using the link "l". The message and the data page are copied to the queue of the receiving process. The content of the data page in the sender's data segment is not modified.

If the receiving process is waiting for a message on the queue, and is higher priority than the sending process, then the sender is suspended.

## ASSEMBLER

```
R0-R7: msg
R8: l
R9: dataPage
KCALL 3
```

## SEE ALSO

Receive, ReceivePage, Send

## NOTES

ErInvLink is returned if the link is not valid.

ErRcvrDead is returned if the receiving process has terminated.

ErQueueFull is returned if the receiver's queue is full, and the message is discarded.

ErNoPageRoom is returned if the receiver's data page pool is full, and the message is discarded.

SetSpecialPID - set a special process ID

## FORMAT

```
FUNCTION SetSpecialPID (genericID : Integer;
                        pID : ProcessID) : Error;
```

## DESCRIPTION

SetSpecialPID is used to register the process ID of certain special system processes, which can then be accessed by other processes. The "pID" of the Volume Manager is registered if "genericID" is 1, while the "pID" of the Directory Manager is registered if "genericID" is 2.

The allocation of the generic identifiers must be coordinated by the processes using them as the Kernel attaches no meaning to them. This interface is subject to change.

## ASSEMBLER

```
R0: genericID
R1: pID
KCALL 43
```

## SEE ALSO

GetSpecialPID, MyID

## NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErBadSpecialPID is returned if the genericID is not valid.

ErInvProc is returned if the process ID is not valid.

ErSpecialPIDInUse is returned if the genericID has already been registered.

SetTimeOfDay - set the system date and time

## FORMAT

```
FUNCTION SetTimeOfDay (time : TimeStamp) : Error;
```

## DESCRIPTION

SetTimeOfDay sets the current system date and time. The "time" is the number of nanoseconds since the start of the year 1900.

## ASSEMBLER

```
R0,R1: time
KCALL 27
```

## SEE ALSO

TimeOfDay

## NOTES

ErNotPriv is returned if the caller is a non-privileged process.

## Suspend - suspend a process

### FORMAT

```
FUNCTION Suspend (pID : ProcessID) : Error;
```

### DESCRIPTION

Suspend makes the process "pID" ineligble to execute, removing it from the ready list.

The process will be suspended until a subsequent Activate call is made to reactivate the process. It is not considered an error to Suspend an already suspended process.

### ASSEMBLER

```
R0: pID
KCALL 17
```

### SEE ALSO

Activate, CreateProcess, Kill

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

Terminate - delete the process of the caller

## FORMAT

```
PROCEDURE Terminate (errorCode : Error);
```

## DESCRIPTION

Terminate is the normal method of deleting the calling process when the process is finished.

The exit code "errorCode" is returned in a message to the parent of the process, notifying the parent of the termination. By convention, the value zero indicates successful completion, while nonzero values indicate some type of error.

## ASSEMBLER

```
R0: errorCode
KCALL 33
```

## SEE ALSO

Kill

## NOTES

This routine never returns to the calling process.

## Test - check if a message is available

### FORMAT

```
FUNCTION Test (q : Queue) : Error;
```

### DESCRIPTION

Test examines the queue "q" to see if a message is available.

The status returned indicates whether the queue is empty, or if the first message in the queue has a data page associated with it.

### ASSEMBLER

```
R8: q
KCALL 5
```

### SEE ALSO

Arm, Receive, ReceivePage, Wait

### NOTES

ErInvQueue is returned if the queue is not valid.

ErQueueEmpty is returned if the queue is empty.

ErNoPageSent is returned if a message is available, and it does not have a data page associated with it.

ErPageSent is returned if a message is available, and it has a data page associated with it.

TimeOfDay - get the system date and time

## FORMAT

```
FUNCTION TimeOfDay : TimeStamp;
```

## DESCRIPTION

TimeOfDay returns the current system date and time. The date and time are encoded as the number of nanoseconds since the start of the year 1900.

Different, increasing values will be returned for each successive call to TimeOfDay from any process in the system.

## ASSEMBLER

```
KCALL 26
R0,R1: TimeOfDay
```

## SEE ALSO

Runtime, SetTimeOfDay

## Trans - translate a virtual address to a real address

FORMAT

```
FUNCTION Trans (virtualAddr : VirtualAddress;
                dirty : Boolean;
            var realAddr : VirtualAddress) : Error;
```

DESCRIPTION

Trans translates a virtual address in the data segment of the calling process into the real address in memory that currently maps the virtual page. The virtual address "virtualAddr" is translated, and the real memory address is returned in "realAddr". The virtual page should first be locked in memory, so that the translation will not later be made invalid by the virtual memory system.

If "dirty" is True, then the modified bit for the virtual page will be set in the VRT, otherwise, it is not changed. The modified bit for the page reflects whether the memory contents have been changed.

ASSEMBLER

```
R0: virtualAddr
R1: dirty
KCALL 32
R0: realAddr
```

SEE ALSO

Fix

NOTES

ErNotPriv is returned if the caller is a non-privileged process.

## Wait - wait for a message

### FORMAT

```
PROCEDURE Wait;
```

### DESCRIPTION

Wait suspends the calling process until a message arrives on an armed queue. When a message arrives on any armed queue, the process will be awakened and the Wait routine will return.

Since no indication is given of which queues received messages since Wait was called, the process should check all armed or unarmed queues using the Test routine. The process should receive all the messages on each queue that is not empty before Wait is called again.

Any messages which arrive while the process is not waiting will cause the next Wait to return immediately.

### ASSEMBLER

```
KCALL 8
```

### SEE ALSO

Arm, Receive, ReceivePage, Test

## WriteProcessState - set the state of a process

### FORMAT

```
FUNCTION WriteProcessState (pID : ProcessID;
                            state : PProcessState) : Error;
```

### DESCRIPTION

WriteProcessState is used to modify the current state of process "pID". The process state includes the general registers, program counter, and the traps word of the process. The priority and the process runtime may also be changed. The "parent", "status" and "msgWait" fields are ignored.

WriteProcessState retrieves a block of information from the data area specified by "state". The ProcessState record is described earlier in this section.

### ASSEMBLER

```
R0: pID
R1: state
KCALL 40
```

### SEE ALSO

ReadProcessState

### NOTES

ErNotPriv is returned if the caller is a non-privileged process.

ErInvProc is returned if the process ID is not valid.

ErBadPCBPointer is returned if the ProcessState record is improperly aligned, or the data structure crosses a page boundary.

# SECTION 4

# RUNTIME LIBRARY INTERFACE

The Runtime Library provides an interface to the Ridge Operating System for user application programs. The Runtime Library supports file system and input/output device access, including file and directory manipulation, plus command process management primitives. In addition, a set of string utilities and time conversion routines are included.

## RUNTIME LIBRARY ORGANIZATION

The Runtime Library is a collection of routines that are linked into a user process, and communicate with the system processes to access their services. The basic Kernel process communication primitives are used to send and receive messages between the user process and the system processes. Most requests are directed to the User Monitor process, which provides an interface to the other system processes. Some requests, such as reading or writing a file or device, are sent directly to a File Manager process or a device driver process.

The Runtime Library routines are called by the user program which supplies the appropriate arguments. A particular routine will convert the arguments into one or more messages that are sent to the appropriate system process, and then the routine will wait for a response message. The user process is suspended until a response message is received. When a response is received, the routine will convert the message back into the appropriate return values and return to the calling program.

The procedural interface supported by the Runtime Library insulates the user program from the details of interprocess communication. In addition, several higher level functions such as buffering of data or string manipulation can be performed in the user process address space, thus contributing to higher performance.

Several of the routines in the Runtime Library use arguments that require variable length sequences of characters to specify file or directory pathnames. These routines use a data structure called a "string" that contains an array of characters and a length field. Strings and routines to manipulate them are described later in this section. Strings that must be passed from one process to another process are generally restricted to fit in a 4096-byte page that is used by the interprocess message primitives.

### Basic File Manipulation

The Runtime Library interface includes several routines that allow access to a "file". A file is simply a sequence of bytes, which may be stored on a secondary storage medium and accessed via the file system, or it may be data exchanged with a peripheral device driver. A file is external to the address space of a process, and must be explicitly read or written to gain access to its contents.

Throughout this section, the word "file" is used to refer to a named collection of data that is managed by the system and accessed from a program via the Runtime Library routines. Thus, a file subsumes the more limited meaning of a storage container within the file system, since it includes data exchanged with a device driver or other process that supports the file manipulation primitives.

The model of a file supported by the basic file manipulation routines is very simple. More sophisticated styles of access are provided by the "stream" file routines described later in this section, or by high-level language statements supported by the various language translators. These higher level abstractions are provided on top of the basic file primitives described here.

A file is a sequence of 8-bit bytes, with no other interpretation placed upon the data by the system. For instance, fixed-size records within a file or the use of text line separators such as the RETURN character are supported by higher level software, and the methods of their implementation are not dictated by the basic file system primitives.

Before any data can be read or written to a file, it must be activated by either the Open or the Create routine, which takes a file name specified as a string. The Open routine takes an existing file, and prepares it for either read-only, write-only , or both read and write access. The Create routine either creates a file if one by that name did not exist, or it erases the previous contents of the file so it can be rewritten. A "link" to the activated file is returned by these routines, and is used in all further accesses to the open file.

A file has a certain size, which is expressed in terms of the number of bytes in the file. The Open routine returns the file size as an argument. The size of a file that represents an interactive device may not be determinable when the file is first accessed, and the device driver may employ some special mechanism for indicating the end of file.

It is not possible to read past the end of the file. The file size may be automatically increased by writing more data at the end of the file. Alternatively, the size of an open file may be extended or truncated by the ChangeFileSize routine.

A file can be read or written using either character or block mode. Some files may only be accessed in one or the other mode, while others may be accessed using either mode. The mode can be determined from a flag that is returned by the Open and Create routines.

Character mode allows single byte transfers, which are usually appropriate for an interactive device such as a terminal that is accessed as a file. Character mode always assumes sequential access to the data, so that for instance, the next character written follows the previous one written to the file. The routines ReadChar and WriteChar are used for character mode access.

Block mode allows transfers of up to 4096 bytes of data in one request, which is oriented towards files implemented on secondary storage such as a disc. Block mode assumes random access to a specified page of the file. The desired page is given as a byte position within the file, and is transferred directly to or from a specified buffer address within the user process data segment. The routines ReadBlock and WriteBlock are used for block mode access.

When no further access is desired to an open file, it should be deactivated by the Close routine. A file can be removed from the file system by the Delete routine, which takes a name in the form of a string. An empty directory (one that contains no file or subdirectory entries) can also be removed by the Delete routine.

## Stream Files

The Runtime Library interface includes several routines that allow access to "stream" files. A stream file helps support the model of files as defined in the language Pascal, acting as a bridge to the basic file handling provided by the Runtime Library.

A stream file is implemented on top of the basic file manipulation primitives described above. Conceptually, a stream file has a read/write cursor associated with it that indicates the next character to be accessed. Normally, access is sequential, with each read or write moving the cursor forward in the file by the number of characters accessed. The Runtime Library routines handle any buffering that must be done to make block-oriented files accessible one character at a time.

Stream files are associated with Pascal variables of type Text (File of Char). The standard Pascal input/output operations on Text variables, including "get", "put", "read", "readln", "write", "writeln", "eof", "eoln" and "page", are implemented using stream files. Refer to the Ridge Pascal Reference Manual for details concerning the standard Pascal input/output operations.

The routine OpenFile is used to associate a Text variable with a stream file. The name of the file is specified as a string, and refers to a file or device driver supported by the Ridge Operating System. The mode of access can be either for reading, writing, appending (writing at the end of the file), or updating (reading and writing). Depending on the mode, the file may be created if it does not exist. A stream file must be opened before any input/output operations are applied to the corresponding Text variable.

The routine FileStatus is used to check for any errors on an open stream file. The status can be checked immediately after an OpenFile call to insure that the file was correctly opened, or it may be checked after an input/output operation to insure that the transfer was completed without error.

The routine PositionFile allows random access to any byte position within a stream file, provided the underlying basic file or device driver supports random access. The read/write cursor can be placed at any byte position relative to the start, end, or current position, thus affecting any further reads or writes.

The SetFileSize routine is used to extend or truncate a stream file to the specified number of bytes. When no further access to a stream file is required, the CloseFile routine should be used to close the file. Any open stream files are automatically closed when a user program terminates normally.

## Directory Manipulation

Several routines in the Runtime Library support directory manipulation. The CreateSpecial routine takes a pathname and creates a directory with that name. The current working directory pathname is retrieved by the GetCurrentDir routine, while ChangeDir changes the current working directory to the given pathname.

The routine LookupName takes a pathname and either returns the mapping of that name to an internal file identifier, or indicates that the pathname is a directory. The internal file identifier can be used to read the file label for that file. The ReadLabel routine returns the contents of a file label, which includes information about the file such as the last time of reference or modification, the ownership and access rights, and the file size.

The ReadDirectory routine takes a pathname of a directory and returns its contents in a standard format. A directory contains entries that map a name to an internal file identifier, or indicate that the entry is the name of a subdirectory.

## Command Process Management

A "command process" provides an environment for a program that is invoked as a command. Most user programs are generally invoked by the Shell command interpreter and run as a command process. The Runtime Library interface includes several routines to manage command processes.

The environment of a command process includes the command arguments accumulated by the invoking process. The GetArgs routine allows a command process to retrieve its command arguments. The command arguments are returned as a vector of strings, where each string represents a single argument to the command. The command process can then interpret its arguments, which may be file names to be operated on or options to control the execution of the program.

The SysExit routine exits back to the system, thus terminating the command process. Any open files are closed, and a exit code is returned to the invoking process. By convention, an exit code value of zero means successful completion, while nonzero values indicate some form of failure.

A command process can be created by the LoadCommand routine. The code file name of the process is specified as a string, and a process ID is returned which is used in all further management of the process. The command process is inactive when first created, and must be activated by the StartCommand routine. The command arguments are supplied with the StartCommand routine, and a flag indicates whether the invoking process should be suspended until the command process terminates. The AbortCommand routine is used to terminate a command process.

The environment of a command process includes a set of aliases or "name equations". A name equation provides a mapping from one string to another string. Name equations can be used as a convenient abbreviation for a file or directory pathname, or they can be used to allow a program to access several different objects by one name without requiring changes to the program. A name equation for a command process is created via the CreateEquate routine, while DeleteEquate deletes a name equation.

## String Manipulation

The Runtime Library contains a set of routines to manipulate "strings". A string is a data structure that contains an arbitrary sequence of characters and a length field that indicates how many characters are in the sequence. Any 8-bit character can appear in a string, and there is no limit on the length of a string, provided of course that it fits in the data segment of a process.

A string is implemented as a pointer to a record, which contains a length field and an array of characters that is indexed from one. Thus, the arguments to the string routines are simply pointers and a new result string can be passed back as a pointer. The precise Pascal definition of a string is given later in this section.

A string data structure is allocated in the heap by the NewString routine. The size of the string is given as the number of characters, and the initial sequence of characters is undefined. The calling program must fill in the desired characters.

The Dispose routine is used to deallocate a string when it is no longer needed. The heap storage management allows strings to be allocated and deallocated in any order.

The ConcatString routine takes two strings as arguments, and returns a new string that is the concatenation of the two strings. CopyOfString returns a new string that is a copy of the argument string. SubString returns a new string that is a substring of the argument string. A substring is specified as the sequence of characters starting at one position in a string and ending at another position.

The OverlayString routine copies the contents of one string onto another, which should be the same length as the source string. CopySubString copies a substring of one string into a specified place in another string.

Two strings may be compared to determine if they have identical contents by the EqualString routine. SearchString searches for a given character in a string, and returns the first position in which it is found. FillString is used to fill a part of a string with a given character.

In order to increase their efficiency, the string routines do little or no checking of input arguments to insure that they are in range for the given function. Instead, the calling program is responsible for checking arguments that might be invalid.

## Time Conversion

The Runtime Library interface provides two routines for conversion of "timestamps". A timestamp is a 64-bit encoding of the number of nanoseconds since the beginning of the year 1900. The system keeps track of the current date and time as a timestamp, and uses timestamps to record when certain events occur, such as the updating of a file.

The EncodeTime routine takes a date and time, specified as several numbers including the year, the month, the day, the hour, the minute and the second, and converts them into a timestamp. The reverse conversion is accomplished by the DecodeTime routine.

## RUNTIME LIBRARY INTERFACE DATA STRUCTURES

The Runtime Library interface routines have arguments that can be defined in terms of several data structures. The following Pascal type definitions describe these data structures, and are assumed in the description of each of the routines found later in this section. Some of the definitions are the same as the ones used for the Kernel, since the underlying support for these data types is provided by the Kernel.

The packing option must be used in Pascal programs that call Runtime Library routines which use types that are affected by the packing option, such as Boolean or Halfword. See the Ridge Pascal Reference Manual for information on packing of data.

```
     Halfword    = 0..65535;
```

A Halfword is 16 bits.

```
     Double      = Set of 0..63;
```

A Double is a 64-bit double word that is normally treated as just a string of bits.

```
     Error       = Integer;
```

An Error is a status that is returned by most interface routines. The value zero indicates successful completion, while nonzero values indicate some failure. Each routine may define specific values to indicate certain errors. Additionally, higher level routines may pass back error values returned by lower level routines.

```
     ProcessID   = Integer;
```

A ProcessID is an identifier for a process which is returned when a process is created. Further manipulation of the process requires the ProcessID.

```
     Link        = Integer;
```

A Link is an identifier for an outbound channel to another process. A Link is returned when a file is opened, and is used for further manipulation of that file.

```
     PageAddress = Integer;
```

A PageAddress contains the virtual address of a 4096-byte page, which must be aligned on a page boundary.

```
     TimeStamp   = Double;
```

A TimeStamp is an encoding of a date and time. It represents the number of nanoseconds since the beginning of the year 1900.

```
     FileID      = Double;
```

A FileID serves as an internal file identifier. A string name maps into a FileID, which is used by the file system to internally specify a file.

```
     String      = ^ StringBody;
     StringBody  = Record
                     length : Integer;
                     chars  : Array [1..1] of Char;
                   end;
```

A String is a pointer to a data structure that contains a sequence of characters. The StringBody contains a "length" field which specifies the number of characters in the sequence, while the "chars" array holds the characters. The sequence can contain an arbitrary number of characters, which are indexed starting at one. For example, if the variable "s" is of type String, then "s^.length" gives the number of characters in "s", while "s^.chars[1]" accesses the first character in "s".

## RUNTIME LIBRARY ROUTINES

The following list of Runtime Library interface routines summarizes their usage, and is grouped according to function. Each routine is described in detail later in this section, where each routine is listed alphabetically in a standard format for easy reference.

### Stream File Manipulation

| | |
|---|---|
| OpenFile | Open a stream file, preparing it for read or write access. |
| CloseFile | Close a stream file. |
| FileStatus | Check the status of an open stream file. |
| PositionFile | Move the read/write cursor within a stream file, placing it at any byte position relative to the start, end, or current position. |
| SetFileSize | Extend or truncate a stream file to the specified number of bytes. |

### Basic File Manipulation

| | |
|---|---|
| Create | Create a new file, or prepare to rewrite an existing file. |
| Delete | Delete a file. An empty directory can also be deleted. |
| Open | Open an existing file for reading or writing. |
| Close | Close an open file. |
| ReadBlock | Read a block of data from a file. Up to 4096 bytes can be read from any page within a file. |
| WriteBlock | Write a block of data to a file. Up to 4096 bytes can be written to any page within a file. |
| ReadChar | Read a single character from a file. |
| WriteChar | Write a single character to a file. |
| ChangeFileSize | Change the size of a file, either extending or truncating the file to the specified number of bytes. |

## Directory Manipulation

| | |
|---|---|
| CreateSpecial | Create a directory or special file. |
| GetCurrentDir | Retrieve the pathname of the current working directory. |
| ChangeDir | Change the current working directory to the given pathname. |
| LookupName | Lookup a pathname to determine its internal identifier in the file system. |
| ReadDirectory | Read the contents of a directory. A directory contains entries that map a name to an internal identifier in the file system. |
| ReadLabel | Retrieve the contents of a file label. A file label contains information about a file such as reference or modification times, ownership and access rights, and file size. |

## Command Process Management

| | |
|---|---|
| GetArgs | Retrieve the command arguments of a process. |
| SysExit | Exit from a process back to the system, thus terminating that process. |
| LoadCommand | Create a command process by specifying the code file name. |
| StartCommand | Activate a command process, supplying its command arguments. |
| AbortCommand | Terminate a command process. |
| CreateEquate | Create a name equation for a command process. |
| DeleteEquate | Delete a name equation. |

## String Manipulation

| | |
|---|---|
| NewString | Create a new string of the specified length. |
| Dispose | Deallocate a string. |
| ConcatString | Concatenate two strings, creating a third string. |

| | |
|---|---|
| CopyOfString | Make a copy of a string, creating a new string. |
| SubString | Make a new string that is a substring of another one. |
| OverlayString | Copy the contents of one string onto another string. |
| CopySubString | Copy a substring of one string into a specified place in another string. |
| EqualString | Test if two strings have identical contents. |
| SearchString | Search a string for a given character. |
| FillString | Fill a string with a given character. |

Time Conversion

| | |
|---|---|
| EncodeTime | Convert a date and time to a timestamp encoding. |
| DecodeTime | Convert a timestamp encoding to a date and time. |

## Error Return Codes

In the detailed description of each routine, error return codes are specified symbolically. An error code always starts with the characters "Er", as in "ErBadFileName". The mapping of these symbols to actual integer values, plus a brief description of each error, is contained in the Appendix.

It is not feasible to list all possible error codes returned by each routine. Some routines may return many different error codes, usually because an error is encountered at a lower level and the error code is passed back through several levels to the caller.

## AbortCommand - abort a command process

### FORMAT

```
FUNCTION AbortCommand (pID : ProcessID) : Error;
```

### DESCRIPTION

AbortCommand is used to kill a command process. Any files left open by the command process will be closed, and other system resources will be freed.

The command process "pID" must have been created by the LoadCommand routine, and can be either active or suspended. A user process that calls AbortCommand must be managed by the same User Monitor that manages the process "pID"; in other words, it is not possible with AbortCommand to kill processes that belong to another user.

An exit code is returned to the invoking process that started the command process via the StartCommand routine. The exit code value is currently undefined when a process is killed by AbortCommand.

### SEE ALSO

LoadCommand, StartCommand

### NOTES

ErBadPID is returned if "pID" is not a valid command process ID.

## ChangeDir - change current working directory

FORMAT

```
FUNCTION ChangeDir (name : String) : Error;
```

DESCRIPTION

ChangeDir changes the current working directory to the pathname "name". The current working directory is the default prefix for pathnames not beginning with "/".

SEE ALSO

GetCurrentDir

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters, while ErNotDirectory is returned if the name is not a directory.

ChangeFileSize - change the size of a file

## FORMAT

```
FUNCTION ChangeFileSize (handle : Link;
                         desiredSize : Integer) : Error;
```

## DESCRIPTION

ChangeFileSize extends or truncates the size of the open file specified by "handle" to the number of bytes specified by "desiredSize". The amount of secondary storage space allocated to the file is changed if necessary.

The file must have been opened for writing.

## SEE ALSO

Create, Open

## NOTES

ErBadLink is returned if "handle" is not a valid open file, while ErNotWritable indicates that the file cannot be written.

## Close - close a file

### FORMAT

```
FUNCTION Close (handle : Link) : Error;
```

### DESCRIPTION

Given a link "handle" to a file as returned by a Create or Open call, Close will close the associated file. Programs which use large numbers of files should use Close when no more access to a file is required, since there is a limit on the number of open files per process. All open files for a process are closed automatically when the process terminates.

Unused secondary storage space may be deallocated, although Close does not change the logical size of a file.

### SEE ALSO

Create, Open

### NOTES

ErBadLink is returned if "handle" is not a valid open file.

## CloseFile - close a stream file

FORMAT

```
PROCEDURE CloseFile (var f : Text);
```

DESCRIPTION

CloseFile closes the stream file associated with "f", which must have been opened by the OpenFile routine. Closing a stream file causes any buffers to be flushed if necessary, and the file becomes inactive.

All open stream files are closed automatically when a program terminates normally, or when the SysExit routine is called.

SEE ALSO

Close, OpenFile, SysExit

## ConcatString - concatenate two strings

### FORMAT

```
FUNCTION ConcatString (s1 : String;
                       s2 : String) : String;
```

### DESCRIPTION

ConcatString creates a new string that is the concatenation of the strings "s1" and "s2". The new string is returned, and neither "s1" nor "s2" are modified.

### SEE ALSO

NewString

CopyOfString - make a copy of a string

## FORMAT

```
FUNCTION CopyOfString (s : String) : String;
```

## DESCRIPTION

CopyOfString creates a new string that is a copy of string "s". The new string is returned, with the same length and same contents as "s". String "s" is not modified.

## SEE ALSO

NewString, OverlayString

## CopySubString - copy a substring into another string

FORMAT

```
PROCEDURE CopySubString (dest : String;
                         dFirst : Integer;
                         dLast : Integer;
                         source : String;
                         sFirst : Integer);
```

DESCRIPTION

CopySubString copies the substring from "source" to the substring in "dest". The characters starting at position "sFirst" in the string "source" are copied to the characters in positions "dFirst" to "dLast" in the string "dest".

SEE ALSO

NewString, SubString

NOTES

No bounds check is made to insure that either of the substrings are completely contained within the strings. No check is made for overlapping substrings when the source and destination are the same string.

## Create - create a file

### FORMAT

```
FUNCTION Create (name : String;
                 accessMode : Halfword;
                 allocSize : Integer;
             var handle : Link;
             var flag : Integer) : Error;
```

### DESCRIPTION

Create tries to create a new file, or prepares to rewrite an existing file. The string "name" represents the pathname of the file.

If the file did not exist, it is given access mode "accessMode", and a file size of zero. If the file did exist, its mode and owner remain unchanged, but the file size is truncated to zero length. In either case, the file will be allocated sufficient storage space to hold "allocSize" bytes.

The file is then opened for both reading and writing, and "handle" will contain a link which is used for subsequent input/output operations on the file.

A value is returned in "flag" that indicates whether the file should be accessed using block mode only (0), character mode only (1), or either mode (2).

### SEE ALSO

Close, Delete, Open

### NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

If the file could not be created because of insufficient secondary storage space, ErVolumeFull is returned.

ErNotWritable is returned if either the file exists and its access mode is not writable, or the file does not exist but the directory in which it is to be created is not writable.

ErCantModifyDir is returned if a directory with the pathname "name" already exists.

## CreateEquate - create a name equation

FORMAT

```
FUNCTION CreateEquate (pID : ProcessID;
                       mapFrom : String;
                       mapTo : String) : Error;
```

DESCRIPTION

CreateEquate creates a name equation for the alias "mapFrom" to the string "mapTo". The name equation is added to the list of name equations maintained by the User Monitor for each command process. When a name is specified in a request to the User Monitor, the name equation list is searched and if the name matches "mapFrom", the string "mapTo" is substituted for the name. The most recent equation for "mapFrom" is used in the case of multiple equations for the same string.

The parameter "pID" specifies to which command process the name equation applies. If the value of "pID" is -1, then the name equation applies to all of the command processes managed by a single User Monitor.

When a command process terminates, its list of name equations is automatically deleted. The list associated with all command processes managed by a single User Monitor can only be shortened by DeleteEquate.

SEE ALSO

DeleteEquate

NOTES

ErBadFileName is returned if the strings are too long, while ErBadPID is returned if "pID" is not a valid command process ID.

CreateSpecial - create a directory or special file

## FORMAT

```
FUNCTION CreateSpecial (name : String;
                        accessMode : Halfword;
                        deviceFlag : Integer) : Error;
```

## DESCRIPTION

CreateSpecial creates a directory whose pathname is given by "name". The directory is given the access mode "accessMode".

The parameter "deviceFlag" must be zero to create a directory. A nonzero value is used to create a device driver, which is not implemented. This interface is subject to change.

## SEE ALSO

Create, Delete

## NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

DecodeTime - convert a timestamp to a date and time

## FORMAT

```
PROCEDURE DecodeTime (time : TimeStamp;
                  var year : Integer;
                  var month : Integer;
                  var day : Integer;
                  var hour : Integer;
                  var minute : Integer;
                  var second : Integer;
                  var millisecond : Integer;
                  var nanosecond : Integer);
```

## DESCRIPTION

DecodeTime converts a timestamp to a set of numbers that represent a date and time. The timestamp "time" is an encoding of the number of nanoseconds since the beginning of the year 1900. DecodeTime converts the timestamp as described below, handling leap years properly.

The "year" will contain a value greater than or equal to 1900, the "month" will contain a value from 1 to 12, and the "day" will contain a value from 1 to 31. The "hour" will contain a value from 0 to 23, and both the "minute" and the "second" will contain a value from 0 to 59. The "millisecond" will contain a value from 0 to 999, and the "nanosecond" will contain a value from 0 to 999999.

## SEE ALSO

EncodeTime

## Delete - delete a file

### FORMAT

```
FUNCTION Delete (name : String) : Error;
```

### DESCRIPTION

Delete removes "name" from the file system. The contents of a file are destroyed, and the secondary storage space is freed.

A directory may be removed only if it is empty.

### SEE ALSO

Create

### NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the name cannot be found.

ErDirNotEmpty is returned if "name" is a directory, and it contains one or more entries.

DeleteEquate - delete a name equation

## FORMAT

```
FUNCTION DeleteEquate (pID : ProcessID;
                       mapFrom : String) : Error;
```

## DESCRIPTION

DeleteEquate removes the name equation for the alias "mapFrom" from the list of name equations maintained by the User Monitor for the command process "pID". If "pID" is the value -1, then the name equation is removed from the list of name equations that applies to all command processes managed by a single User Monitor. If multiple name equations exist for "mapFrom", then only the most recent mapping will be deleted.

When a command process terminates, its list of name equations is automatically deleted. DeleteEquate is used to either delete a name equation before using CreateEquate to change a mapping, or to delete a name equation from the list for all command processes.

## SEE ALSO

CreateEquate

## NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErBadPID is returned if "pID" is not a valid command process ID.

ErNoEquate is returned if "mapFrom" cannot be found.

Dispose - deallocate a string

## FORMAT

```
PROCEDURE Dispose (s : String);
```

## DESCRIPTION

Dispose deallocates the data structure associated with string "s", and makes its space on the heap available for reuse. The heap storage management allows strings to be allocated and deallocated in any order.

## SEE ALSO

NewString

## NOTES

Unpredictable results will occur if "s" had never been allocated, or if it had already been disposed.

EncodeTime - convert a date and time to a timestamp

## FORMAT

```
PROCEDURE EncodeTime (year : Integer;
                      month : Integer;
                      day : Integer;
                      hour : Integer;
                      minute : Integer;
                      second : Integer;
                      millisecond : Integer;
                      nanosecond : Integer;
                  var time : TimeStamp);
```

## DESCRIPTION

EncodeTime converts a set of numbers that represent a date and time to a timestamp. The timestamp "time" is an encoding of the number of nanoseconds since the beginning of the year 1900. EncodeTime converts the set of numbers as described below, handling leap years properly.

The "year" should contain a value greater than or equal to 1900, the "month" should contain a value from 1 to 12, and the "day" should contain a value from 1 to 31. The "hour" should contain a value from 0 to 23, and both the "minute" and the "second" should contain a value from 0 to 59. The "millisecond" should contain a value from 0 to 999, and the "nanosecond" should contain a value from 0 to 999999.

## SEE ALSO

DecodeTime

## NOTES

An input parameter that is out of range results in a timestamp that is equal to zero.

## EqualString - compare two strings for equality

### FORMAT

```
FUNCTION EqualString (s1 : String;
                      s2 : String) : Boolean;
```

### DESCRIPTION

EqualString compares the two strings "s1" and "s2" for equality. The value True is returned if both strings have exactly the same length and contents, otherwise the value False is returned. The contents are compared character by character, using the underlying character code values, thus upper and lower case characters are not identical.

### SEE ALSO

NewString

FileStatus - check the status of a stream file

## FORMAT

```
FUNCTION FileStatus (var f : Text) : Error;
```

## DESCRIPTION

FileStatus checks the status of the open stream file associated with "f". The value returned is zero if no errors have occurred during any operations on the stream file.

The status may be tested immediately after an OpenFile call to check that the file was correctly opened, or it may be checked after an input/output operation to insure that the transfer was successfully completed. Reaching the end of file is not considered an error.

## SEE ALSO

OpenFile

## NOTES

ErNotOpen is returned if the stream file is not open, while ErFileStatus is returned if any error has occurred.

FillString - fill a string with a character

## FORMAT

```
PROCEDURE FillString (s : String;
                      first : Integer;
                      last : Integer;
                      ch : Char);
```

## DESCRIPTION

FillString fills a substring within string "s" with the character "ch". The characters from the position "first" to the position "last" are all given the value of "ch".

## SEE ALSO

NewString

## NOTES

No bounds check is made to insure that the substring is completely contained within the string.

## GetArgs - get command arguments

### FORMAT

```
FUNCTION GetArgs (var argc : Integer) : PStringVector;
```

### DESCRIPTION

GetArgs is used by a command process to retrieve its command arguments from the User Monitor. The command arguments are typically file names to be operated on, or options to control the execution of the program. The invoking process, usually the Shell, accumulates the command arguments and passes them to the User Monitor via StartCommand when a command process is started.

GetArgs sets the argument count into "argc" and returns a pointer to an array of strings, as described by the following Pascal type definitions:

```
PStringVector = ^ StringVector;
StringVector  = Array [0..0] of String;
```

The array is indexed from zero, and actually contains "argc"+1 elements, where the last element is the value Nil.

By convention, the string at position zero is the name of the command process, and the other strings are the command arguments in sequence. Thus "argc" is always at least one.

### SEE ALSO

StartCommand

### NOTES

The strings returned by GetArgs should not be deallocated via Dispose since they are not allocated by NewString.

GetCurrentDir - get the name of the current working directory

## FORMAT

```
FUNCTION GetCurrentDir : String;
```

## DESCRIPTION

GetCurrentDir returns a string which is the pathname of the current working directory.

## SEE ALSO

ChangeDir

LoadCommand - create a command process

FORMAT

```
FUNCTION LoadCommand (name : String;
                      var pID : ProcessID) : Error;
```

DESCRIPTION

LoadCommand creates a command process, using the file "name" as the executable code segment. The code file is found using the standard search order. If the name starts with "/", then that exact pathname is used. Otherwise, a pathname is constructed by appending "name" first to the current working directory, then the directory "/bin", and finally the directory "/usr/bin".

If a code file is found, a data and a queue segment are allocated, and an inactive process is created. A process ID is returned in "pID", which is used for further management of the command process.

SEE ALSO

AbortCommand, StartCommand

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the file can not be found, while ErBadFileType is returned if it is not a regular file.

## LookupName - lookup a file name

### FORMAT

```
FUNCTION LookupName (name : String;
                     var fType : Integer;
                     var fID : FileID) : Error;
```

### DESCRIPTION

LookupName takes a pathname "name" and determines its mapping in the file system. If the name is a regular file, then "fType" will contain the value 1, and "fID" will contain the internal file identifier.

If the name is a directory, then "fType" will contain the value 0. The contents of "fID" for a directory are interpreted as four 16-bit values in sequence according to the following Pascal type definitions:

```
ownerID   = Halfword;
groupID   = Halfword;
protect   = Halfword;
linkCount = Halfword;
```

The "ownerID" and "groupID" fields indicate the owner of the directory. The "protect" field is the protection bits, or access mode, for the directory. The "linkCount" field represents the number of aliases, or links, to the directory from other directories in the file system.

### SEE ALSO

ReadDirectory, ReadLabel

### NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErFileNotFound is returned if the file or directory can not be found.

NewString - create a new string

## FORMAT

```
FUNCTION NewString (length : Integer) : String;
```

## DESCRIPTION

NewString creates a new string, with enough space to hold "length" characters. A data structure is allocated on the heap, with the length field filled in, and undefined values for the sequence of characters. A pointer to this structure is returned by NewString, and the calling program can fill in the desired characters.

When the string is no longer required, its space should be deallocated by the Dispose routine.

## SEE ALSO

Dispose

## Open - open a file

### FORMAT

```
FUNCTION Open (name : String;
               mode : Integer;
           var handle : Link;
           var flag : Integer;
           var fileSize : Integer) : Error;
```

### DESCRIPTION

Open tries to open an existing file, where the string "name" represents the pathname of the file. The file will be opened for reading ("mode" is 0), writing ("mode" is 1), or both reading and writing ("mode" is 2).

If the file is opened successfully, "handle" will contain a link which is used for subsequent input/output operations on the file. The file will be positioned at its beginning.

A value is returned in "flag" that indicates whether the file should be accessed using block mode only (0), character mode only (1), or either mode (2).

The value returned in "fileSize" is the number of bytes in the file. The size of a file that represents a device driver may not be determinable when the file is first accessed, and will be zero in this case.

### SEE ALSO

Close, Create

### NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters.

ErOpenMode is returned if "mode" is not 0, 1, or 2.

ErFileNotFound is returned if the file can not be found.

## OpenFile - open a stream file

### FORMAT

```
PROCEDURE OpenFile (var f : Text;
                        name : String;
                        mode : Char);
```

### DESCRIPTION

OpenFile is used to associate a stream file with the variable "f", which can then be used in standard Pascal input/output operations or can be a parameter to other stream file routines. The string "name" specifies the pathname of the file. The character "mode" can be one of four different letters (upper or lower case) that indicate how the file should be opened, as described below.

The value "r" indicates the file should be opened for reading only, and therefore must exist.

The value "w" indicates the file should be opened for writing only, and is either created if it did not exist, or truncated to zero length if it did exist.

The value "a" indicates the file should be appended to, which is similar to writing only. The file is created if it did not exist. If the file exists, it is not truncated and the read/write cursor is positioned at the end of file.

The value "u" indicates that the file should be opened for update, which permits both reading and writing. The file is created if it did not exist, and the read/write cursor is positioned at the beginning of the file.

### SEE ALSO

CloseFile, Create, FileStatus, Open

### NOTES

The FileStatus routine should be used to determine if a stream file was opened successfully. A stream file may not be opened correctly if the file name is not valid, does not exist or cannot be accessed according to the mode, or if too many open files already exist.

OverlayString - copy one string onto another

## FORMAT

```
PROCEDURE OverlayString (dest : String;
                         source : String);
```

## DESCRIPTION

OverlayString copies the characters of the "source" string onto the previous contents of the "dest" string, and sets the length of "dest" to that of "source". The "source" string is not modified.

## SEE ALSO

CopyOfString, NewString

## NOTES

Unpredictable results may occur if the destination string was not at least as long as the source string.

PositionFile - move the read/write cursor of a stream file

## FORMAT

```
FUNCTION PositionFile (var f : Text;
                           offset : Integer;
                           origin : Integer) : Integer;
```

## DESCRIPTION

PositionFile moves the read/write cursor of the stream file associated with "f". The next input or output operation on the stream file will occur at the new position, where position 0 is the first byte of the file.

The new position becomes the byte position determined from the signed value in "offset" and the value of "origin". If "origin" is 0, then the offset is from the beginning of the file; if "origin" is 1, then the offset is relative to the current position; and if "origin" is 2, then the offset is relative to the end of the file. The resulting byte position in the file is returned.

A successful PositionFile always clears the end of file status.

## SEE ALSO

OpenFile, SetFileSize

## NOTES

The value -1 is returned to indicate an error. An error may occur if "origin" is not 0, 1, or 2, if the stream file is not open or does not support block mode or random access, or if the position would be beyond the end of file.

ReadBlock - read a block of a file

## FORMAT

```
FUNCTION ReadBlock (handle : Link;
                     bufAddr : PageAddress;
                     fileCursor : Integer;
                     length : Integer;
                 var actual : Integer) : Error;
```

## DESCRIPTION

ReadBlock is used to read a block of data from the file specified by "handle". The "handle" is a file link returned from a successful Create or Open call, and it represents a file that must allow block mode access.

The block of data from the file at the byte position specified by "fileCursor" is transferred to the buffer address in the user process data segment specified by "bufAddr", which must be page-aligned.

The amount of data to be read is specified in "length", and the amount actually transferred is returned in "actual", which can be from 0 to 4096 bytes.

## SEE ALSO

Create, Open, ReadChar, WriteBlock, WriteChar

## NOTES

ErBadLink is returned if "handle" is not a valid open file.

ErNotAligned is returned if "bufAddr" is not page-aligned, that is, an address that is not a multiple of 4096, while ErBadBlockLength is returned if "length" is greater than 4096.

ErNotReadable is returned if the file was not opened to allow reading.

ErEOF is returned if an attempt is made to read a block past the end of file.

## ReadChar - read a character

### FORMAT

```
FUNCTION ReadChar (handle : Link;
               var ch : Char) : Error;
```

### DESCRIPTION

ReadChar reads a single character from the file specified by "handle". The "handle" is a file link returned from a successful Create or Open call, and it represents a file that must allow character mode access.

The character is returned in "ch", and can be any 8-bit value.

### SEE ALSO

Create, Open, ReadBlock, WriteBlock, WriteChar

### NOTES

ErBadLink is returned if "handle" is not a valid open file.

ErNotReadable is returned if the file was not opened to allow reading.

ErEOF is returned if an attempt is made to read a character past the end of file.

## ReadDirectory - read the contents of a directory

### FORMAT

```
FUNCTION ReadDirectory (name : String;
                        firstRequest : Boolean;
                        dirPage : PDirectoryPage;
                    var numEntries : Integer;
                    var anyMore : Boolean) : Error;
```

### DESCRIPTION

ReadDirectory is used to read the contents of a directory. The directory contents are returned in a standard format, which contains entries that map a name to an internal file identifier, or indicate that an entry is the name of a subdirectory.

Each call to ReadDirectory returns only one 4096-byte page of information. The entries are returned in alphabetical order so that multiple requests can be made, each time specifying a different place in the alphabetical list to start returning more entries. The parameters "firstRequest" and "anyMore" are used as described below to make multiple requests, thus enabling a large directory to be read.

The first call to ReadDirectory for a directory whose pathname is "name" should have the parameter "firstRequest" set to True. The number of valid entries for the returned directory page is returned in "numEntries". If more information exists in the directory than can be returned in a single response, then the parameter "anyMore" will be set to True upon return, otherwise "anyMore" will be set to False.

If "anyMore" is True, then another call to ReadDirectory should be made with "firstRequest" set to False. The parameter "name" should be extended to include the directory name and the name of the last entry returned, separated by a "/". The returned information will start with the next alphabetical entry.

ReadDirectory places a block of directory information into the data area specified by "dirPage" in the following format:

```
PDirectoryPage = ^ DirectoryPage;
DirectoryPage  = Array [0..127] of DirectoryEntry;
DirectoryEntry = Record
                   name  : Array [1..16] of Char;
                   fType : Integer;
                   fID   : FileID;
                 end;
```

The page may contain up to 128 entries, starting with entry number zero. Each entry has a "name" field, which is 1 to 16 characters with blanks filled at the end. The "fType" field has the value 1 if the entry is a regular file, and the "fID" field contains an internal file identifier in this case.

The "fType" field has the value 0 if the entry is a directory. The contents of the "fID" field for a directory are interpreted as four 16-bit values in sequence according to the following Pascal type definitions:

```
ownerID   = Halfword;
groupID   = Halfword;
protect   = Halfword;
linkCount = Halfword;
```

The "ownerID" and "groupID" fields indicate the owner of the directory. The "protect" field is the protection bits, or access mode, for the directory. The "linkCount" field represents the number of aliases, or links, to the directory from other directories in the file system.

SEE ALSO

CreateSpecial, LookupName, ReadLabel

NOTES

ErBadFileName is returned if the name is too long, or contains invalid characters, while ErNotDirectory is returned if the name is not a directory.

ReadLabel - read a file label

FORMAT

```
FUNCTION ReadLabel (fID : FileID;
                    lab : PFileLabel) : Error;
```

DESCRIPTION

ReadLabel is used to read the contents of the file label for the file specified by the internal file identifier "fID". The file label contains information about the file that is maintained by the file system.

ReadLabel places a block of file label information into the data area specified by "lab" in the following format:

```
PFileLabel = ^ FileLabel;
FileLabel  = Record
               internalCreate : TimeStamp;
               createTime     : TimeStamp;
               refTime        : TimeStamp;
               modTime        : TimeStamp;
               ownerID        : Halfword;
               groupID        : Halfword;
               protect        : Halfword;
               linkCount      : Halfword;
               fileSize       : Integer;
               uType          : Integer;
             end;
```

The "internalCreate" field contains the time that the file was actually created. The "createTime" field contains the time that the file was logically created, which may be different than the "internalCreate" time if the file is a copy of another file, for instance. The "refTime" field contains the last time that the file was referenced, that is, closed after being opened for reading or writing. The "modTime" field contains the last time that the file was modified, that is, closed after being opened for writing.

The "ownerID" and "groupID" fields indicate the owner of the directory. The "protect" field is the protection bits, or access mode, for the file. The "linkCount" field represents the number of name mappings, or links, to the file from directories in the file system.

The "fileSize" field maintains the size of the file in bytes. The "uType" field contains a file type value maintained by the system.

## SEE ALSO

LookupName, ReadDirectory

## NOTES

ErBadFileID is returned if "fID" is not a valid internal file identifier.

SearchString - search for a character in a string

## FORMAT

```
FUNCTION SearchString (s : String;
                       ch : Char) : Integer;
```

## DESCRIPTION

SearchString searches the string "s" for the first occurrence of the character value "ch". If "ch" is found, then its position in the sequence of characters is returned. Zero is returned if "ch" can not be found.

## SEE ALSO

NewString

## SetFileSize - change the size of a stream file

### FORMAT

```
FUNCTION SetFileSize (var f : Text;
                          desiredSize : Integer) : Error;
```

### DESCRIPTION

SetFileSize extends or truncates the size of the stream file associated with "f" to the number of bytes specified by "desiredSize". The amount of secondary storage space allocated to the file is changed if necessary.

The stream file must have been opened with a mode that allows writing.

If the stream file is truncated in front of the current read/write cursor, then the read/write cursor is positioned to the new end of file.

### SEE ALSO

OpenFile, ChangeFileSize, PositionFile

### NOTES

ErNotWritable is returned if the file is not writable.

## StartCommand - start a command process executing

### FORMAT

```
FUNCTION StartCommand (pID : ProcessID;
                        args : PArgPage;
                        wait : Boolean) : Error;
```

### DESCRIPTION

StartCommand is used to start execution of a command process. The command process "pID" must have been created by the LoadCommand routine; StartCommand activates "pID", supplying its command arguments.

The command arguments are passed in a 4096-byte page to the User Monitor, which transmits them to the command process when it calls GetArgs. The parameter "args" points to a page which contains the arguments in the following format:

```
PArgPage = ^ ArgPage;
ArgPage  = Record
             argc : Integer;
             strings : Array [0..0] of StringBody;
           end;
```

The argument page contains an argument count and zero or more strings packed sequentially. The strings consist of a length field followed by the sequence of characters, with the length fields aligned on word (4-byte) boundaries. By convention, the argument count is at least one, with the first string being the command name the process was invoked with.

If "wait" is True, then the invoking process is suspended until the command process terminates. In this case, the exit code of the command process is returned as the value of the StartCommand routine. If "wait" is False, then the invoking process is not suspended and no indication is given when the command process terminates.

### SEE ALSO

AbortCommand, GetArgs, LoadCommand, SysExit

### NOTES

ErBadPID is returned if "pID" is not a valid command process ID.

ErCantStart is returned if the command process cannot be properly activated.

SubString - make a copy of a substring

## FORMAT

```
FUNCTION SubString (s : String;
                    first : Integer;
                    last : Integer) : String;
```

## DESCRIPTION

SubString creates a new string that is a substring of string "s". The characters from position "first" through position "last" in "s" are copied into the new string, which is then returned. String "s" is not modified.

## SEE ALSO

CopySubString, NewString

## NOTES

No bounds check is made to insure that the substring is completely contained within the string.

SysExit - exit back to the system

## FORMAT

```
PROCEDURE SysExit (errorCode : Error);
```

## DESCRIPTION

SysExit exits back to the system, thus terminating the calling command process. Any open stream files are closed before the process is terminated.

The exit code "errorCode" is returned to the invoking process which started the command process via StartCommand. By convention, the value zero indicates successful completion, while nonzero "errorCode" values indicate different errors defined by the command process.

## SEE ALSO

CloseFile, StartCommand

## NOTES

This routine never returns to its caller.

## WriteBlock - write a block of a file

FORMAT

```
FUNCTION WriteBlock (handle : Link;
                     bufAddr : PageAddress;
                     fileCursor : Integer;
                     length : Integer;
                 var actual : Integer) : Error;
```

DESCRIPTION

WriteBlock is used to write a block of data to the file specified by "handle". The "handle" is a file link returned from a successful Create or Open call, and it represents a file that must allow block mode access.

The block of data from the buffer address in the user process data segment specified by "bufAddr", which must be page-aligned, is transferred to the file at the byte position specified by "fileCursor".

The amount of data to be written is specified in "length", and the amount actually transferred is returned in "actual", which can be from 0 to 4096 bytes.

The file size is increased by any bytes which extend past the current end of file.

SEE ALSO

Create, Open, ReadBlock, ReadChar, WriteChar

NOTES

ErBadLink is returned if "handle" is not a valid open file.

ErNotAligned is returned if "bufAddr" is not page-aligned, that is, an address that is not a multiple of 4096, while ErBadBlockLength is returned if "length" is greater than 4096.

ErNotWritable is returned if the file was not opened to allow writing.

ErBadFileCursor is returned if an attempt is made to write a block beyond the current end of file, which would leave a gap in the file.

WriteChar - write a character

## FORMAT

```
FUNCTION WriteChar (handle : Link;
                    ch : Char) : Error;
```

## DESCRIPTION

WriteChar writes a single character to the file specified by "handle". The "handle" is a file link returned from a successful Create or Open call, and it represents a file that must allow character mode access.

The character "ch" that is written can be any 8-bit value.

## SEE ALSO

Create, Open, ReadBlock, ReadChar, WriteBlock

## NOTES

ErBadLink is returned if "handle" is not a valid open file.

ErNotWritable is returned if the file was not opened to allow writing.

# APPENDIX

## ERROR RETURN CODES

The system interface routines may return various error codes when a routine is unable to successfully complete the desired function. The following list contains the error values in decimal, the symbolic name for the error starting with the characters "Er", and a short explanation of what the error indicates.

| | |
|---|---|
| 1 ErQueueFull | A message can not be sent because the queue of the receiving process is full. |
| 2 ErRcvrDead | A message can not be sent because the receiving process has terminated. |
| 3 ErInvLink | The specified link is invalid or not open. |
| 4 ErNotPageBound | A page address is not aligned on a 4096-byte boundary. |
| 5 ErNoPageRoom | There is not enough room in the queue of the receiving process to accept a message with a data page attached. |
| 6 ErNotPriv | The calling process is not privileged. |
| 7 ErPageSent | The next message to be received has a data page associated with it. |
| 8 ErNoPageSent | The next message to be received does not have a data page associated with it. |
| 9 ErQueueEmpty | No messages have been received in a queue. |
| 14 ErEOF | An attempt has been made to read past the end of file, or write beyond the end of file, leaving a gap. |
| 19 ErTooManyLinks | Too many links have been requested when initializing a queue segment. |
| 20 ErTooManyQueues | Too many queues have been requested when initializing a queue segment. |
| 21 ErTooManyPages | Too many pages have been requested when initializing a queue segment. |

| | |
|---|---|
| 22 ErTooManyMessages | Too many messages have been requested when opening a queue. |
| 23 ErInvProc | The specified process identifier is invalid, or the process has terminated. |
| 24 ErInvQueue | The specified queue is invalid or not open. |
| 25 ErNoProcBlock | A process cannot be created because there are not enough unused process control blocks. |
| 26 ErNoFreeSegments | A process cannot be created because there are not enough unused segment identifiers. |
| 27 ErNoFreeQueues | A queue cannot be opened because there are not enough unused queues. |
| 28 ErNoFreeLinks | A link cannot be opened because there are not enough unused links. |
| 29 ErInvDevice | The specified device number is invalid. |
| 30 ErDeviceInUse | The device number is already in use by another process. |
| 31 ErBadPCBPointer | The process control block address is not properly aligned, or causes the block to cross a page boundary. |
| 32 ErBadSpecialPID | The specified special process identifier is invalid. |
| 33 ErPageNotReserved | A page cannot be unlocked because it is not currently locked. |
| 34 ErSpecialPIDInUse | The special process identifier is already in use by another process. |
| 35 ErInvSeg | The specified segment selector is invalid. |
| 102 ErNoFreePages | Not enough memory pages would be left if the requested page was locked. |
| 201 ErBadFileID | The specified internal file identifier is invalid. |
| 206 ErVolumeIndexFull | A file cannot be created because there are not enough unused file labels. |
| 208 ErVolumeFull | A file cannot be created or extended because there is not enough room on the volume. |
| 303 ErBadFileName | The specified file name is too long or contains invalid characters. |

| | |
|---|---|
| 304 ErBadFileType | The type field in a directory for the specified name is invalid. |
| 306 ErFileNotFound | A file with the specified name cannot be found. |
| 308 ErNotDirectory | The specified name is not a directory. |
| 313 ErDirNotEmpty | The directory cannot be deleted because it is not empty. |
| 401 ErBadPID | The specified command process identifier is invalid. |
| 402 ErOpenMode | The mode given when opening a file is invalid. |
| 403 ErBadLink | The specified file link is invalid or not open. |
| 404 ErNoEquate | The specified name equation does not exist. |
| 405 ErCantStart | The command process cannot be activated or otherwise started. |
| 407 ErCantModifyDir | A directory cannot be modified using the given operation. |
| 408 ErBadFileSpace | The space allocation of a file is not one contiguous sequence of bytes starting from position zero. |
| 410 ErNotReadable | The specified file does not allow read access. |
| 411 ErNotWritable | The specified file does not allow write access. |
| 412 ErBadFileCursor | The specified address of a page to be read or written is not within the file. |
| 413 ErBadBlockLength | The length of a block to be read or written to a file is less than 0 or greater than 4096 bytes long. |
| 499 ErNotImplemented | A feature has not been implemented yet. |
| 500 ErNotOpen | A stream file is not currently open. |
| 501 ErFileStatus | An error has occurred on an open stream file. |
| 502 ErNotAligned | A page address is not aligned on a 4096-byte boundary. |