OPERATING SYSTEM REFERENCE MANUAL

for the LISA™

CONTENTS

Chapter 1
INTRODUCTION
1.1 The Main Functions 1-3
1.2 Using the OS Functions 1-3
1.3 The File System 1-4
1.4 Process Management 1-6
1.5 Memory Management 1-7
1.6 Exceptions and Events 1-7
1.7 Interprocess Communication 1-8
1.8 Using the OS Interface 1-8
1.9 Running Programs Under the OS 1-8
1.10 Writing Programs That Use the OS 1-8

Chapter 2
THE FILE SYSTEM
2.1 File Names 2-3
2.2 The Working Directory 2-4
2.3 Devices 2-5
2.4 Storage Devices 2-5
2.5 The Volume Catalog 2-6
2.6 Labels 2-6
2.7 Logical and Physical End Of File 2-6
2.8 File Access 2-7
2.9 Pipes 2-8
2.10 File System Calls 2-9

Chapter 3
PROCESSES
3.1 Process Structure 3-4
3.2 Process Hierarchy 3-4
3.3 Process Creation 3-5
3.4 Process Control 3-5
3.5 Process Scheduling 3-6
3.6 Process Termination 3-7
3.7 A Process Handling Example 3-7
3.8 Process System Calls 3-9

Chapter 4
MEMORY MANAGEMENT
4.1 Data Segments 4-3
4.2 The Logical Data Segment Number 4-3
4.3 Shared Data Segments 4-4
4.4 Private Data Segments 4-4
4.5 Code Segments 4-4
4.6 Swapping 4-5
4.7 Memory Management System Calls 4-5

Chapter 5
EXCEPTIONS AND EVENTS
5.1 Exceptions 5-3
5.2 System Defined Exceptions 5-4
5.3 Exception Handlers 5-4
5.4 Events 5-7
5.5 Event Channels 5-7
5.6 The System Clock 5-12
5.7 Exception Management System Calls 5-12
5.8 Event Management System Calls 5-18
5.9 Clock System Calls 5-28

Chapter 6
CONFIGURATION
6.1 Configuration System Calls 6-3

Appendix A
OPERATING SYSTEM INTERFACE

Appendix B
SYSTEM RESERVED EXCEPTION NAMES

Appendix C
SYSTEM RESERVED EVENT TYPES

Appendix D
OPERATING SYSTEM ERROR MESSAGES

Appendix E
FS_INFO FIELDS

INDEX

TABLES

2-1 Device Control Functions Required Before Using a Device 2-27
2-2 Device_Control Output Functional Groups 2-28
2-3 Dccode Mneumonics 2-30
2-4 Device Information 2-32
2-5 Disk Hard Error Codes 2-34

FIGURES

2-1 Disk Hard Error Codes 2-34
2-2 The Reslationship of COMPACT and TRUNCATE 2-38
3-1 Process Address Space Layout 3-4
3-2 Process Tree 3-5
5-1 Stack at Exception Handler Invocation 5-6

PREFACE

The Contents of this Manual

This manual describes the Operating System service calls that are available to Pascal and assembler programs. It is written for experienced Pascal programmers, and does not explain elementary terms and programming techniques. We assume that you have read the *Lisa Owner's Guide* and *Workshop User's Guide for the Lisa* and are familiar with your Lisa system.

Chapter 1 is a general introduction to the Operating System.

Chapter 2 describes the File System and the available file system calls. This includes a description of the interprocess communication facility, pipes, and the Operating System calls that allow processes to use pipes.

Chapter 3 describes the calls available to control processes, and also describes the structure of processes.

Chapter 4 describes how processes can control their use of available memory.

Chapter 5 describes the use of events and exceptions to control process synchronization. It also describes the use of the system clock.

Chapter 6 describes the calls you can use to find out information about the configuration of the system.

Appendix A contains the source text of **SYSCALL**, the unit that contains the type, procedure, and function definitions discussed in this manual.

Appendix B contains a list of system-defined exception names.

Appendix C contains a list of system-defined event names.

Appendix D contains a list of error codes that may be produced by the calls documented in this manual.

Type and Syntax Conventions

Bold face type is used in this manual to distinguish programming keywords and constructs from English text. For example, **FLUSH** is the name of a system call. System call names are also capitalized in this manual, although Pascal does not distinguish between lower and upper case characters. *Italics* are used to indicate new terms that are to be explained.

Chapter 1
INTRODUCTION

1.1 The Main Functions 1-3
1.2 Using the OS Functions 1-3
1.3 The File System 1-4
1.4 Process Management 1-6
1.5 Memory Management 1-7
1.6 Exceptions and Events 1-7
1.7 Interprocess Communication 1-8
1.8 Using the OS Interface 1-8
1.9 Running Programs Under the OS 1-8
1.10 Writing Programs That Use the OS 1-8

INTRODUCTION

The Operating System (OS) provides an environment in which multiple processes can coexist, with the ability to communicate and share data. It provides a file system for I/O and information storage, and handles exceptions (software interrupts) and memory management.

1.1 The Main Functions

The OS has four main functional areas: the File System, Process Management, Memory Management, and event and exception handling.

The File System provides input and output. The File System accesses devices, volumes, and files. Each object, whether a printer, disk file, or any other type of object, is referenced by a pathname. Every I/O operation is performed as an uninterpreted byte stream. Using the File System, all I/O is device independent. The File System also provides device specific control operations.

A process is an executing program and its associated data. Several processes can execute concurrently by multiplexing the processor between them. These processes can be broken into segments which are automatically swapped into memory as needed.

Memory managment routines handle data segments. A data segment is a file that can be placed in memory and accessed directly.

Exceptions and events are process communication constructs provided by the OS. An event is a message sent from one process to another, or from a process to itself, that is delivered to the receiving process only when the process asks for that event. An exception is a special type of event that forces itself on the receiving process. There is a set of system defined exceptions (errors), and programs can define their own. System errors such as division by zero are examples of system defined exceptions. You can use the system calls provided to define any exceptions you want.

All four of these areas are described further later in this chapter.

1.2 Using the OS Functions

Both built in language features and explicit OS system calls can access OS routines to perform desired functions. For example, the Pascal **writeln** procedure is a built in feature of the language. The code to execute a writeln is supplied in IOSPASLIB, the pascal run time support routines library. This code is added to the program when the program is linked. The code provided calls OS File System routines to perform the desired output.

You can also call OS routines explicitly. This is usually done when the language does not provide the operation you want. OS routines allow Pascal programs, for example, to create new processes, which could not otherwise be done, since Pascal does not have any built in process handling functions.

All calls to the OS are synchronous, which means they do not return until the operation is complete. Each call returns an error code to indicate if anything went wrong during the operation. Any non zero value indicates an error or warning. Negative error codes indicate warnings.

1.3 The File System

The File System performs all I/O as uninterpreted byte streams. These byte streams can go to files on disk or to other devices, such as a printer or an alternative console. In all cases, the device or file has a File System name. Except for device control functions, the File System treats devices and files in the same way.

The File System allows sharing of all types of objects.

The File System provides for naming objects (devices, files, etc.). A name in the File System is called a *pathname*. A complete pathname consists of a directory name and a file name. The file name is meaningful only for storage devices (devices that store byte streams for later use, such as disks).

Each process has a working directory associated with it. This allows you to reference objects with an incomplete pathname. To access an object in the working directory, just give its file name. To access an object in a different directory, give its complete pathname.

Before a device can be accessed, it must be mounted. Devices can be mounted using the Preference tool, or by using the MOUNT call (see Chapter 2 of this manual). If the device is a storage device, the mount operation makes a *volume name* available. A volume name is a logical name for a disk, and is saved on the disk itself. The mount operation logically connects the volume to the system, so that the files on the volume may be accessed. The volume name can replace a device name in a pathname used to access an object on the disk. The volume name allows you to access a file with the same pathname no matter where the drive is actually connected.

If a device is specified in the configuration list (created by the Preference tool) and it is physically connected to the Lisa, then the device can be accessed. There are some operations that can be performed on unmounted (unconfigured) devices. Two examples are DEVICE_CONTROL calls (see Chapter 2 of this manual) and scavanging. Logically mounting a volume on a device makes file access to the volume possible. For storage devices, a volume is an actual magnetic medium that can contain recorded files. For non-storage devices, volumes and files are

concepts used to maintain a uniform interface. Files on non-storage devices (such as printers) do not store data, but acts as "ports" for performing I/O to the devices.

The basic operations provided by the File System are as follows:

```
mount and unmount - make a volume accessible/inaccessible
open and close - make an object accessible/inaccessible
read and write - transfer information to and from an
                 object
device control functions -  control device specific
                            functions
```

Some operations apply only to storage devices:

```
allocate and deallocate -   specify size of an object
manipulate catalog - controls naming of objects and
                creation and destruction of objects
manipulate attributes - look at or change the
                        characteristics of the object
```

In addition to the data in an object, the object itself has certain characteristics. These are called its *attributes*. They include such information as the length and creation date of a file. Calls are available to access the attributes of any File System object. In addition to the system defined attributes, objects on a storage device can have a *label*. The label is available for programs to store information that they can interpret.

Non-storage devices (such as printers) are accessed with a limited set of operations. They must be mounted and opened before they can be accessed. Sequential read and/or write operations are available as appropriate for the device. Device control functions are available to perform any device specific functions needed. The file name portion of the complete pathname for a non-storage device is not used by the File System, although you do have to provide one when you open the device.

For storage devices, the same sequential read and write operations are valid as for non-storage devices. Storage devices also must be mounted, and particular files opened, before the files can be used. They have appropriate device control functions available.

When writing to a disk file, space for the file is allocated as needed. Space for a file does not need to be contiguous, and in some cases this automatic allocation can result in a fragmented file, which may slow file access. To insure rapid access, you can pre-allocate space for the file. Pre-allocating the file also ensures that the process will not run out of space on the disk.

Four types of objects can be stored on storage devices. These are files, pipes, data segments, and event channels. Files, already discussed, are simply arrays of stored data. Pipes are objects provided for inter-process communication. Data

segments are special cases of files that are loaded into memory along with program code. Event channels are pipes with a specialized structure imposed by the system.

1.4 Process Management

A process is an executing program and its associated data. Several processes can exist at one time, and they appear to run simultaneously because the CPU is multiplexed among them. The scheduler decides what process should use the CPU at any one time. It uses a generally non-preemptive scheduling algorithm. This means that a process will not lose the CPU unless it blocks.

A process can lose the CPU when one of the following happens:

- The process calls an Operating System procedure or function
- The process references one of its code segments that is not currently in memory

If neither of these occur, the process will not lose the CPU.

Every process is started by another process. The newly started process is called the *son process*. The process that started it is called its *father process*. The resulting structure is a tree of processes (See Figure 3-2).

When any process terminates, all its son processes (and their descendants) are also terminated.

When the OS is booted, it starts a shell process. The shell process then starts any other processes desired by the user.

Every newly created process has the same system-standard attributes and capabilities. These can be changed by using system calls.

Any processes can suspend, activate, or kill any other process for which the global ID is known, as long as the other process does not protect itself.

The memory accesses of an executing process are restricted to its own memory address space. Processes can communicate with other processes by using shared files, pipes, event channels, or shared data segments.

A process can be in one of three states: ready, running, or blocked. A *ready* process is waiting for the scheduler to select it to run. A *running* process is currently using the CPU to execute its code. A *blocked* process is waiting for some event, such as the completion of an I/O operation. It will not be scheduled until the event occurs, at which point it becomes ready. A *terminated* process has finished executing.

Each process has a priority from 1 to 255. The higher the number, the higher the priority of the process. Priorities 226 to 255 are reserved for system processes.

The scheduler always runs the ready process with the highest priority. A process can change its own priority, or the priority of any other process, while it is executing.

1.5 Memory Management

Memory managment is concerned with what is in physical memory at any one time. Each process can use up to 128 segments. Each segment can contain up to 128 Kbytes. These segments are of two types: code segments and data segments. The total amount of memory used by any one process can exceed the available RAM of the Lisa. The Operating System will swap code segments in and out of memory as they are needed. To aid the Operating System in swapping data segments, calls are provided to give programs the ability to define which data segments must be in memory while a particular part of the program is executing.

You have control of how your program is divided up. For executable code segments, you use the segmentation commands of the Pascal compiler to break the program in pieces.

In addition to residing in memory, data segments can be stored permanantly on disk. They can be accessed with calls similar to File System calls. This allows you to use a data segment as a direct access file -- a file that is accessed as part of your memory space.

Calls are provided for making, killing, opening, and closing data segments. You can also change the size of a data segment and set its access mode to read only or read write. In addition, you can make a permanent disk copy of the contents of a data segment at any time. Other calls give you ability to force the contents of the data segment to be swapped into main memory, so they can be accessed by your process.

1.6 Exceptions and Events

An exception is an unexpected condition in the execution of a process (an interrupt). An event is a message from another process.

An exception can be generated by either the system or an executing program. System exceptions are generated by various sorts of errors such as divide by zero, illegal instruction, or illegal address. System exception handlers are supplied that terminate the process. You can write your own exception handlers for any of these exceptions if you want to try to recover from the error.

User exceptions can be declared, and exception handlers written to process them. Your program can then signal this new exception.

Events are messages sent from one process to another. They are sent through event channels.

A process that wants to receive a message from an event channel executes a call to wait for an event on that channel. This will give it the next message, if one exists, or block the process until a message arrives.

If a process wants to know when an event arrives, but does not want to wait for it, it can use a call event channel. This is set up by associating a user exception with the event channel when it is opened. The Operating System will then invoke the corresponding user exception handler whenever a message arrives in the event channel.

1.7 Interprocess Communication

There are four methods for interprocess communication. These are: shared files, pipes, event channels, and shared data segments.

Shared files are used for high volume transfers of information. It is necessary to coordinate the processes somehow to prevent them from overwriting each other's information.

Pipes are used for communication between processes with an uninterpreted byte stream. The pipe mechanism provides for the needed synchronization because a process will block if it is trying to read from an empty pipe or write to a full one. A read from a pipe consumes the information, so it is no longer available. Only one process can read from a given pipe.

Event channels are similar to pipes, except they transmit short, structured messages instead of uninterpreted bytes.

A shared data segment can be used to transmit a large amount of data rapidly. Having a shared data segment means that this data segment is in the memory address space of all the processes that want to use it. All the processes can then directly read and write information in the data segment. It is necessary to provide some sort of synchronization to keep one process from overwriting another's information.

1.8 Using the OS Interface

The interface to all the system calls is provided in the unit SYSCALL. This unit can be used to provide access to the calls. See the *Workshop User's Guide for the Lisa* for more information on using SYSCALL.

1.9 Running Programs Under the OS

Programs can be written and run by using the Workshop.

1.10 Writing Programs That Use the OS

You can write a program that calls OS routines to perform needed functions. This program USES the unit SYSCALL, then calls the routines needed.

Chapter 2
THE FILE SYSTEM

2.1 File Names 2-3
2.2 The Working Directory 2-4
2.3 Devices 2-5
2.4 Storage Devices 2-5
2.5 The Volume Catalog 2-6
2.6 Labels 2-6
2.7 Logical and Physical End Of File 2-6
2.8 File Access 2-7
2.9 Pipes 2-8
2.10 File System Calls 2-9
MAKE_FILE 2-10
MAKE_PIPE 2-10
KILL_OBJECT 2-12
UNKILL_FILE 2-13
RENAME_ENTRY 2-14
LOOKUP 2-15
INFO 2-18
SET_FILE_INFO 2-19
OPEN 2-20
CLOSE_OBJECT 2-21
READ_DATA 2-22
WRITE_DATA 2-22
READ_LABEL 2-25
WRITE_LABEL 2-25
DEVICE_CONTROL 2-26
ALLOCATE 2-36
COMPACT 2-37
TRUNCATE 2-38
FLUSH 2-39
SET_SAFETY 2-40
SET_WORKING_DIR 2-41
GET_WORKING_DIR 2-41
RESET_CATALOG 2-42
GET_NEXT_ENTRY 2-42
MOUNT 2-43
UNMOUNT 2-43

THE FILE SYSTEM

The File System provides device independent I/O, storage with access protection, and uniform file naming conventions.

Device independence means that all I/O is performed in the same way, whether the ultimate destination (or source) is disk storage, another program, a printer, or anything else. In all cases, I/O is performed to or from *files*, although those files are often also devices, data segments, or programs.

Every file is an uninterpreted stream of eight-bit bytes.

A file that is stored on a block structured device, such as a disk, is listed in a *catalog* (also called a *directory*) and has a name. For each such file the catalog contains an entry describing the file's attributes, including the length of the file, its position on the disk, and the last backup copy date. Arbitrary application-defined information can be stored in an area called the *file label*. Each file has two associated measures of length, the *Logical End of File (LEOF)* and the *Physical End of File (PEOF)*. The LEOF is a pointer to the last byte that has meaningful data. The PEOF is a count of the number of blocks allocated to the file. The pointer to the next byte to be read or written is called the *file marker*.

Since I/O is device independent, application programs do not have to take account of the physical characteristics of a device. When the I/O is to or from a disk, or any other block structured device, programs can make I/O requests in whole-block increments, which improves program performance.

All input and output is synchronous in that the I/O requested is performed before the call returns. The actual I/O, however, is asynchronous, in that processes may block when performing I/O. (See Section 3.5, Process Scheduling, for more information on blocking.)

To reduce the impact of an error, the file system maintains distributed, redundant information about the files on storage devices. Duplicate copies of critical information are stored in different forms and in different places on the media. All the files are able to identify and describe themselves, and there are usually several ways to recover lost information. The scavenger utility is able to reconstruct damaged catalogs from the information stored with each file.

2.1 File Names

All the files known to the Operating System at a particular time are organized into catalogs. Each disk volume has a catalog that lists all the files on the disk.

Any object catalogued in the file system can be named by specifying the volume on which the file resides and the file name. The names are separated by the character "-". Because the top catalog in the system has no name, all complete pathnames begin with "-".

For example,

```
-LISA-FORMAT.TEXT names a file on a volume named LISA.
```

The file name can contain up to 32 characters. If a longer name is specified, the name is truncated to 32 characters. Accesses to sequential devices use an arbitrary dummy filename that is ignored but must be present in the pathname. For example, the serial port pathname

```
-RS232B
```

is insufficient, but

```
-RS232B-XYZ
```

is accepted, even though the -XYZ portion is ignored. Certain device names are predefined:

RS232A	Serial Port A
RS232B	Serial Port B
PARAPORT	Parallel Port
SLOTxCHANy	Serial ports, where x is 1, 2, or 3 and y is 1 or 2
MAINCONSOLE	**writeln** and **readln** device
ALTCONSOLE	**writeln** and **readln** device
UPPER	Upper Diskette drive (Drive 1)
LOWER	Lower Diskette drive (Drive 2)
BITBKT	Bit bucket: data is thrown away when directed here

See Chapter 6, Configuration for more information on device names.

Upper and lower case are not significant in pathnames: 'TESTVOL' is the same object as 'TestVol'. Any ASCII character is legal in a pathname, including the non-printing characters and blank spaces. (However, use of ASCII 13, a RETURN, in a pathname is strongly discouraged.)

2.2 The Working Directory

It is sometimes inconvenient to specify a complete pathname, especially when working with a group of files in the same volume. To alleviate this problem, the operating system maintains the name of a working directory for each process. When a pathname is specified without a leading "-", the name refers to an object in the working directory. For example, if the working directory is -LISA the name FORMAT.TEXT refers to the same file as -LISA-FORMAT.TEXT. The default working directory name is the name of the boot volume directory.

You can find out what the working directory is with GET_WORKING_DIR. You can change to a new working directory with SET_WORKING_DIR.

2.3 Devices

Device names follow the same conventions as other file names. Attributes like baud rate are controlled by using the DEVICE_CONTROL call with the appropriate pathname.

Each device has a permanently assigned priority. From highest to lowest the priorities are:

```
Power on/off button
Serial Port A (RS232A)
Serial Port B (RS232B, the leftmost port)
I/O Slot 1
I/O Slot 2
I/O Slot 3
Keyboard, mouse, battery powered clock
10 ms system timer
CRT vertical retrace interrupt
Parallel Port
Diskette 1 (UPPER)
Diskette 2 (LOWER)
Video Screen
```

The device driver associated with a device contains information about the device's physical characteristics such as sector size and interleave factors for disks.

2.4 Storage Devices

On storage devices, such as disk drives, the File System reads or writes file data in terms of pages. A page is the same size as a block. Any access to data in a file ultimately translates into one or more page accesses. When a program requests an amount of data that does not fit evenly into some number of pages, the File System reads the next highest number of whole pages. Similarly, data is actually written to a file only in whole page increments.

A file does not need to occupy contiguous pages. The File System keeps track of the locations of all the pages that make up a file.

Each page on a storage device is self-identifying, and the page descriptor is stored with the page contents to reduce the destructive impact of an I/O error.

The eight components of the page descriptor are:

Version number
Volume identifier
File identifier
Amount of data on the page
Page name
Page position in the file
Forward link
Backward link

Each volume has a *Medium Descriptor Data File (MDDF)* which describes the various attributes of the medium such as its size, page length, block layout, and the size of the boot area. The MDDF is created when the volume is initialized.

The File System also maintains a record of which pages on the medium are currently allocated, and a catalog of all the files on the volume. Each file contains a set of file hints, which describe and point to the actual file data.

2.5 The Volume Catalog

On a storage device, the volume catalog provides access to the files. The catalog is itself a file that maps user names into the internal file identifiers used by the Operating System. Each catalog entry contains a variety of information about each file including:

name
type
internal file number and address
size
date and time created, last modified, and last accessed
file identifier
safety switch

The safety switch is used to avoid accidental deletions. While the safety switch is on, the file cannot be deleted. The other fields are described under the LOOKUP file system call.

The catalog can be located anywhere on the medium.

2.6 Labels

An application can store its own information about a file in an area called the *file label*. The label allows an application to keep the file data separate from information maintained about the file. Labels can be used for any object in the file system. The maximum label size is 128 bytes. I/O to labels is handled separately from file data I/O.

2.7 Logical and Physical End of File

A file contains some number of bytes of data recorded in some number of physical pages. Additional pages might be allocated to the file, which do not

contain any file data. There are, therefore, two measures of the end of the file called the logical and physical end of file. The Logical End of File (LEOF) is a pointer to the last stored byte that has meaning to the application. The Physical End of File (PEOF) is a count of the number of pages allocated to the file.

In addition, each open file has a pointer associated with it, called the *file marker*, that points to the next byte in the file to be read or written. When the file is opened, the file marker points to the first byte (byte number 0). The file marker can be positioned implicitly or explicitly using the read and write calls. For example, when a program writes to a file opened with Append access, the file marker is automatically positioned to the end of the file before new data is written. The file marker cannot be positioned past LEOF, except by a write operation that appends data to a file.

When a file is created, an entry for it is made in the catalog specified in its pathname, but no space is allocated for the file itself. When the file is opened by a process, space can be allocated explicitly by the process, or automatically by the Operating System. If a write operation causes the file marker to be positioned past the LEOF marker, LEOF, and PEOF, if necessary, are automatically extended. The new space is contiguous if possible.

2.8 File Access

The File System provides a device independent byte stream interface. As far as an applications program is concerned, a specified number of bytes is transferred either relative to the file marker or at a specified byte location in the file. The physical attributes of the device or file are not important to the application, except that devices that do not support positioning can only perform sequential operations, and programs can improve performance somewhat by taking advantage of a device's physical characteristics.

Programs can request any amount of data from a file. The actual I/O, however, is performed in whole-page increments, when devices are block structured. Therefore, programs can optimize I/O with such devices by setting the file marker on a page boundary and making I/O requests in whole-page increments.

A file can be open for access simultaneously by more than one process. All requests to write to the file are completed before any other access to the file is permitted. When one process writes to a file the effect of that write operation is immediately available to all other processes reading the file. The other processes may, however, have accessed the file in an earlier state. Data already obtained by a program are not changed. The programmer must insure that processes maintain a consistent view of a shared file.

When you open a file, you specify the kind of access allowed on the file. When the file is opened, the Operating System allocates a file marker for the calling process and a run-time identification number called the *refnum*. The process must use the refnum in subsequent calls to refer to the file. Each operation using the refnum affects only the file marker associated with that refnum.

Processes can share the same file marker. In this access mode (global_ refnum) each process uses the same refnum for the file. When a process opens a file in global access mode, the refnum it gets back can be passed to any other process, and used by any process. Note that any number of processes can open a file with global_ refnum, but each time the OPEN call is used a different refnum is produced. Each of those refnums can be passed to other processes, and each process using a particular refum shares the same file marker with other processes with the same refum. Processes using different refnums, however, always have different file markers, whether or not those refums were obtained with global_refnum.

A file can also be opened in private mode, which specifies that no other OPEN calls are to be allowed for that file. A file can be opened with global_refnum and private, which opens the file for global access, but allows no other process to open that file. By using this call, processes can control which other processes have access to a file. The opening process passes the global refnum to any other process that is to have access, and the system prevents other processes from opening the file.

Programmers should be aware that processes using global access may not be able to make any assumptions about the location of the file marker from one access to the next.

2.9 Pipes

Because the Operating System supports multiple processes, a mechanism is provided for interprocess communication. This mechanism is called a *pipe*. Pipes are very similar to the other objects in the file system -- they are named according to the same rules, and they can have labels.

As with a file, a pipe is a byte stream. With a pipe, however, information is queued in a first-in-first-out manner. Also, a pipe can have only one reader at a time, and once data is read from a pipe it is removed from the pipe.

A pipe can only be accessed in sequential mode. Although only one process can read data from a pipe, any number of processes can write data into it. Because the data read from the pipe is consumed, the file marker is always at zero. If the pipe is empty and no processes have it open for writing, EOF (End Of File) is returned to the reading process. If any process does have it open for writing, the reading process is suspended until enough data to satisfy the call arrives in the pipe, or until all writers close the pipe.

When a pipe is created, its size is 0 bytes. Unlike with ordinary files, the initializing program must allocate space to the pipe before trying to write data into it. To avoid deadlocks between the reading process and the writers, the Operating System does not allow a process to read or write an amount of data greater than half the physical size of the pipe. For this reason, you should allocate to the pipe twice as much space as the largest amount of data in any planned read or write operation.

A pipe is actually a circular buffer with a read pointer and a write pointer. All writers access the pipe through the same write pointer. Whenever either pointer reaches the 'end' of the pipe, it wraps back around to the first byte. If the read pointer catches up with the write pointer, the reading process blocks until data are written or until all the writers close the pipe. Similarly, if the write pointer catches up with the read pointer, a writing process blocks until the pipe reader frees up some space or until the reader closes the pipe. Because pipes have this structure, there are certain restrictions on some operations when dealing with a pipe. These restrictions are discussed with the relevant file system calls.

Processes can never make read or write requests bigger than half the size of the pipe because that the Operating System always fully satisfies each read or write request before returning to the program. In other words, if a process asks for 100 bytes of data from a pipe, the Operating System waits until there are 100 bytes of data in the pipe, and then completes the call. Similarly, if a process tries to write 100 bytes of data into a pipe, the Operating System waits until there is room for the full 100 bytes before writing anything into the pipe. If processes were allowed to make write or read requests for greater than half of a particular pipe, it would be possible for a reader and a writer to deadlock, with neither having room in the pipe to satisfy its requests.

2.10 File System Calls

This section describes all the Operating System calls that pertain to the file system. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in the file system calls:

```
Pathname = STRING[Max_Pathname];(* Max_Pathname = 255 *)
E_Name = STRING[Max_Ename];    (* Max_EName = 32 *)
Accesses = (DRead, DWrite, Append, Private, Global_refnum);
MSet = SET OF Accesses;
IoMode = (Absolute, Relative, Sequential);
```

The fs_info record and its associated types are described under the LOOKUP call. The Dctype record is described under the DEVICE_CONTROL call.

```
MAKE_FILE  (Var Ecode:Integer;
            Var Path:Pathname;
            Label_size:Integer)

MAKE_PIPE  (Var Ecode:Integer;
            Var Path:Pathname;
            Label_size:Integer)

          Ecode:      Error indication
          Path:       Name of new object
          Label_size: Number of bytes for the object's label
```

MAKE_FILE and MAKE_PIPE create the specified type of object with the given name. If the pathname does not specify a directory name (more specifically, if the pathanme does not begin with a dash), the working directory is used. Label_size specifies the initial size in bytes of the label that the application wants to maintain for the object. It must be less than or equal to 128 bytes. The label can grow to contain up to 128 bytes no matter what its initial size. Any error indication is returned in Ecode.

The example below checks to see whether the specified file exists before opening it.

```
CONST FileExists = 890;
VAR FileRefNum,ErrorCode:INTEGER;
    FileName:PathName;
    Happy:BOOLEAN;
    Response:CHAR;
BEGIN
 Happy:=FALSE;
 WHILE NOT Happy DO
  BEGIN
   REPEAT                    (*get a file name*)
    WRITE('File name: ');
    READLN(FileName);
   UNTIL LENGTH(FileName)>0;
   MAKE_FILE(ErrorCode,FileName,0);(*no label for this file*)
   IF (ErrorCode<>0) THEN        (*does file already exist?*)
   IF (ErrorCode=FileExists) THEN (*yes*)
    BEGIN
     WRITE(FileName,' already exists. Overwrite? ');
     READLN(Response);
     Happy:=(Response IN ['y','Y']);       (*go ahead and
                                            overwrite*)
    END
```

```
    ELSE WRITELN('Error ',ErrorCode,' while creating file.')
    ELSE Happy:=TRUE;
  END;
 OPEN(ErrorCode,FileName,FileRefNum,[Dwrite]);
END;
```

```
KILL_OBJECT (Var Ecode:Integer;
             Var Path:Pathname)
```

```
Ecode:    Error indicator
Path:     Name of object to be deleted
```

KILL_OBJECT deletes the object given in Path from the file system. Objects with the safety switch on cannot be deleted. If a file or pipe is open at the time of the KILL_OBJECT call, its actual deletion is postponed until it has been closed by all processes that have it open. During this period no new processes are allowed to open it. The object to be deleted need not be open at the time of the KILL_OBJECT call. A KILL_OBJECT call can be reversed by UNKILL_FILE, as long as the object is a file and is still open.

The following program fragment deletes files until carriage return is typed:

```
CONST FileNotFound=894;
VAR FileName:PathName;
    ErrorCode:INTEGER;
BEGIN
 REPEAT
  WRITE('File to delete: ');
  READLN(FileName);
  IF (FileName<>'') THEN
   BEGIN
    KILL_OBJECT(ErrorCode,FileName);
    IF (ErrorCode<>0) THEN
     IF (ErrorCode=FileNotFound) THEN
      WRITELN(FileName,' not found.')
     ELSE WRITELN('Error ',ErrorCode,' while deleting file.')
    ELSE WRITELN(FileName,' deleted.');
   END
 UNTIL (FileName='');
END;
```

```
UNKILL_FILE  (Var Ecode:Integer;
                  Refnum:Integer;
              Var New_name:e_name)
```

```
Ecode:    Error indicator
Refnim:   Refnum of the killed and open file
New_name:New name for the file being restored
```

UNKILL_FILE reverses the effect of KILL_OBJECT, as long as the killed object is a file that is still open. A new catalog entry is created for the file with the name given in New_name. New_name is not a full pathname: the resurrected file remains in the same directory.

```
RENAME_ENTRY  (Var Ecode:Integer;
               Var Path:Pathname;
               Var Newname:E_Name)
```

```
Ecode:   Error indicator
Path:    Object's old name
Newname: Object's new name
```

RENAME_ENTRY changes the name of an object in the file system. Newname can not be a full pathname. The name of the object is changed, but the object remains in the same directory. The following program fragment changes the file name of FORMATTER.LIST to NEWFORMAT.TEXT.

```
VAR OldName:PathName;
    NewName:E_Name;
    ErrorCode:INTEGER
BEGIN
    OldName:='-LISA-FORMATTER.LIST';
    NewName:='NEWFORMAT.TEXT';
    RENAME_ENTRY(ErrorCode,OldName,NewName);
END;
```

The file's new full pathname is '-LISA-NEWFORMAT.TEXT'.

Volume names can be renamed by specifying only the volume name in Path. Here is a sample program fragment which changes a volume name. Note that the leading dash (-), given in Oldname, is not given in Newname.

```
VAR OldName:PathName;
    NewName:E_Name;
    ErrorCode:INTEGER
BEGIN
    Oldname:='-thomas';
    Newname:='stearns';
    RENAME_ENTRY(Errorcode,Oldname,Newname);
END;
```

```
LOOKUP (Var Ecode:Integer;
        Var Path:Pathname;
        Var Attributes:Fs_Info)

  Ecode:      Error indicator
  Path:       Object to lookup
  Attributes: Information returned about path
```

LOOKUP returns information about an object in the file system. For devices and mounted volumes, call LOOKUP with a pathname that names the device or volume without a file name component:

```
DevName:='-UPPER';          (* Diskette drive 1 *)
LOOKUP(ErrorCode, devname, InfoRec);
```

If the device is currently mounted and is block structured, the record fields of Attributes contain meaningful values; otherwise, some values are undefined.

The fs_info record is defined as follows. The meanings of the information fields are given in Appendix E.

```
fs_info = RECORD
          name:   e_name;
          devnum: INTEGER;
CASE OType:info_type OF
          device_t, volume_t:
              (iochannel: INTEGER
              devt:       devtype;
              slot_no:    INTEGER;
              fs_size:    LONGINT;
              vol_size:   LONGINT;
              blockstructured,
              mounted:    BOOLEAN;
              opencount:  LONGINT;
              privatedev,
              remote,
              lockeddev:  BOOLEAN;
              mount_pending,
              unmount_pending: BOOLEAN;
              volname,
              password:   e_name;
              fsversion,
              volid,
              volnum:     INTEGER;
```

```
                blocksize,
                datasize,
                clustersize,
                filecount: INTEGER;(*Number of files on vol*)
                freecount: LONGINT;(*Number of free blocks *)
                DTVC,       (* Date Volume Created *)
                DTVB,       (* Date Volume last Backed up *)
                DTVS:LONGINT;(* Date Volume last scavanged *)
                Machine_id,
                overmount_stamp,
                master_copy_id: LONGINT;
                privileged,
                write_protected: BOOLEAN;
                master,
                copy,
                scavenge_flag: BOOLEAN);
            object_t: (
                size: LONGINT; (*actual no of bytes written*)
                psize: LONGINT; (*physical size in bytes*)
                lpsize: INTEGER; (*Logical page size in bytes*)
                ftype:    filetype;
                etype:    entrytype;
                DTC,                    (* Date Created *)
                DTA,                    (* Date last Accessed *)
                DTM,                    (* Date last Mounted *)
                DTB:      LONGINT;      (*Date last Backed up *)
                refnum:   INTEGER;
                fmark:    LONGINT;      (* file marker *)
                acmode:   mset;         (* access mode *)
                nreaders,   (* Number of readers *)
                nwriters,   (* Number of writers *)
                nusers:   INTEGER;      (* Number of users *)
                fuid:     uid;      (* unique identifier *)
                eof,                  (* EOF encountered? *)
                safety_on,            (* safety switch setting *)
                kswitch:  BOOLEAN;(* has file been killed? *)
                private,(* File opened for private access? *)
                locked, (* Is file locked? *)
                protected:BOOLEAN);(* File copy protected? *)
END;

Uid = INTEGER;
Info_Type = (device_t, volume_t, object_t);
Devtype = (diskdev, pascalbd, seqdev, bitbkt, non_io);
```

```
Filetype  = (undefined, MDDFFile, rootcat, freelist,
             badblocks, sysdata, spool, exec, usercat,
             pipe, bootfile, swapdata, swapcode, ramap,
             userfile, killedobject);
Entrytype = (emptyentry, catentry, linkentry, fileentry,
             pipeentry, ecentry, killedentry);
```

The EOF field of the fs_info record is set after an attempt to read more bytes than are available from the file marker to the logical end of the file, or after an attempt to write when no disk space is available. If the file marker is at the twentieth byte of a twenty-five byte file, for example, you can read up to 5 bytes without setting EOF, but if you try to read 6 bytes, the file system gives you only 5 bytes of data, and EOF is set.

The following program reports how many bytes of data a given file has:

```
VAR InfoRec:Fs_Info;(*information returned by LOOKUP and
                      INFO*)
    FileName:PathName;
    ErrorCode:INTEGER;
BEGIN
  WRITE('File: ');
  READLN(FileName);
  LOOKUP(ErrorCode,FileName,InfoRec);
  IF (ErrorCode<>0) THEN
    WRITELN('Cannot lookup ',FileName)
  ELSE
    WRITELN(FileName,' has ',InfoRec.Size,' bytes of data.');
END;
```

```
INFO (Var Ecode:Integer;
          Refnum:Integer;
      Var RefInfo:Fs_Info)
```

Ecode:	Error indicator
Refnum:	Reference number of object in file system
Refinfo:	Information returned about refnum's object

INFO serves a function similar to that of **LOOKUP**, but is applicable only to objects in the file system which are open. The definition of the Fs_Info record is given under **LOOKUP** and in Appendix A.

```
SET_FILE_INFO ( Var Ecode:Integer;
                    Refnum:Integer;
                    Fsi:Fs_Info)
```

Ecode:	Error indicator
Refnum:	Reference number of object in file system
Fsi:	New information about the object

SET_FILE_INFO changes the status information associated with the given object. This call works in exactly the opposite way that LOOKUP and INFO work, in that the status information is given by your program to SET_FILE_INFO. The fsi argument is the same type of information record as that returned by LOOKUP and INFO. The object must be open at the time this call is made.

The following fields of the information report may be changed:

- file_scavanged
- file_closed by_OS
- file_left_open
- user_type
- user_subtype

```
OPEN (Var Ecode:Integer;
      Var Path:Pathname;
      Var Refnum:Integer;
          Manip:MSet)
```

```
Ecode:     Error indicator
Path:      Name of object to be opened
Refnum:    Reference number for object
Manip:     Set of access types
```

The OPEN call opens an object so that it can be read or written to. When you call OPEN, you specify the set of accesses that will be allowed on that file or sequential device. The available access types are:

- Dread -- Allows you to read any of the file
- Dwrite -- Allows you to write anywhere in the file (replaces existing data)
- Append -- Allows you to add on to the end of the file
- Private -- Prevents other processes from opening the file
- Global_refnum -- Creates a refnum that can be passed to other processes

Note that you can give any number of these modes simultaneously. If you give dwrite and append in the same OPEN call, dwrite access will be used. See Section 2.8 for more information on global_refnum and private access modes.

If the object opened already exists and the process calls WRITE_DATA without having specified append access, the object can be overwritten. The Operating System does not create a temporary file and wait for the CLOSE_OBJECT call before deciding what to do with the old file.

An object can be opened by two separate processes (or more than once by a single process) simultaneously. If the processes write to the file without using a global refnum, they must coordinate their file accesses so as to avoid overwriting each other's data.

Pipes cannot be opened for dwrite access. You must use append if you want to write into the pipe.

```
CLOSE_OBJECT (Var Ecode:Integer;
              Refnum:Integer)
```

```
Ecode:  Error indicator
Refnum: Reference number of object to be closed.
```

If refnum is not global, CLOSE_OBJECT terminates any use of Refnum for I/O operations. A FLUSH operation is performed automatically and the file is saved in its current state. If Refnum is a global refnum, and other processes have the file open, Refnum remains valid for these processes, and other processes can still access the file using Refnum.

The following program fragment opens a file, reads 512 bytes from it, then closes the file.

```
TYPE Byte=-128..127;
VAR  FileName:PathName;
     ErrorCode,FileRefNum:Integer;
     ActualBytes:LongInt;
     Buffer:ARRAY[0..511] OF Byte;
BEGIN
  OPEN(ErrorCode,FileName,FileRefNum,[DRead]);
  IF (ErrorCode>0) THEN
    WRITELN('Cannot open ',FileName)
  ELSE
    BEGIN
      READ_DATA(ErrorCode,
                FileRefNum,
                ORD4(@Buffer),
                512,
                ActualBytes,
                Sequential,
                0);
      IF (ActualBytes<512) THEN
        WRITE('Only read ',ActualBytes,' bytes from
              ',FileName);
      CLOSE_OBJECT(ErrorCode,FileRefNum);
    END;
END;
```

```
READ_DATA (Var Ecode:Integer;
               Refnum:Integer;
               Data_Addr:LongInt;
               Count:LongInt;
           Var Actual:LongInt;
               Mode:IoMode;
               Offset:LongInt);

WRITE_DATA (Var Ecode:Integer;
                Refnum:Integer;
                Data_Addr:Longint;
                Count:LongInt;
            Var Actual:LongInt;
                Mode:IoMode;
                Offset:LongInt);

        Ecode:     Error indicator
        Refnum:    Reference number of object for I/O
        Data_Addr: Address of data (source or destination)
        Count:     Number of bytes of data to be transferred
        Actual:    Actual number of bytes transferred
        Mode:      I/O mode
        Offset:    Offset (absolute or relative modes)
```

READ_DATA reads information from the device, pipe, or file specified by Refnum, and WRITE_DATA writes information to it. Data_Addr is the address for the destination or source of Count bytes of data. The actual number of bytes transferred is returned in Actual.

Mode can be absolute, relative, or sequential. In absolute mode, offset specifies an absolute byte of the file. In relative mode, it specifies a byte relative to the file marker. In sequential mode, the offset is ignored (it is assumed to be zero) and transfers occur relative to the file marker. Sequential mode (which is a special case of relative mode) is the only allowed access mode for reading or writing data in pipes or sequential (non-disk) devices. Non-sequential modes are valid only on devices that support positioning. The first byte is numbered 0.

If a process attempts to write data past the physical end of file on a disk file, the Operating System automatically allocates enough additional space to contain the data. This new space, may not be contiguous with the previous blocks. You can use the ALLOCATE call to insure that any newly allocated blocks are located next to each other, although that will not insure that they are located near the rest of the file.

READ_DATA from a pipe that does not contain enough data to satisfy count suspends the calling process until the data arrives in the pipe if any other process has that pipe open for writing. If there are no writers, the end of file indication (error 848) is returned by Ecode. Because pipes are circular, WRITE_DATA to a pipe with insufficient room suspends the calling process (the writer) until enough space is available (until the reader has consumed enough data), if there is a reader. If no process has the pipe open for reading and there is not enough space in the pipe, the end of file indication (848) is returned in Ecode.

NOTE

READ_DATA from the MAINCONSOLE or ALTCONSOLE devices must specify count=1.

The following program copies a file. Note that you must supply the correct location for Syscall in the second line of the program.

```
PROGRAM CopyFile;
USES (*Syscall.Obj*) SysCall;
TYPE Byte=-128..127;
VAR  OldFile,NewFile:PathName;
     OldRefNum,NewRefNum:INTEGER;
     BytesRead,BytesWritten:LONGINT;
     ErrorCode:INTEGER;
     Response:CHAR;
     Buffer:ARRAY [0..511] OF Byte;
BEGIN
  WRITE('File to copy: ');
  READLN(OldFile);
  OPEN(ErrorCode,OldFile,OldRefNum,[DRead]);
  IF (ErrorCode>0) THEN
  BEGIN
    WRITELN('Error ',ErrorCode,' while opening ',OldFile);
    EXIT(CopyFile);
  END;
  WRITE('New file name: ');
  READLN(NewFile);
  MAKE_FILE(ErrorCode,NewFile,0);
  OPEN(ErrorCode,NewFile,NewRefNum,[DWrite]);
```

```
  REPEAT
    READ_DATA( ErrorCode,
               OldRefNum,
               ORD4(@Buffer),
               512,BytesRead,Sequential,0);
    IF (ErrorCode=0) AND (BytesRead>0) THEN
      WRITE_DATA (ErrorCode,
                  NewRefNum,
                  ORD4(@Buffer),
                  BytesRead,BytesWritten,Sequential,0);
  UNTIL (BytesRead=0) OR (BytesWritten=0) OR
  (ErrorCode>0);
  IF (ErrorCode>0) THEN
    WRITELN('File copy encountered error ',ErrorCode);
  CLOSE_OBJECT(ErrorCode,NewRefNum);
  CLOSE_OBJECT(ErrorCode,OldRefNum);
END.
```

```
READ_LABEL (Var Ecode:Integer;
            Var Path:Pathname;
                Data_Addr:Longint;
                Count:LongInt;
            Var Actual:LongInt)

WRITE_LABEL (Var Ecode:Integer;
             Var Path:Pathname;
                 Data_Addr:Longint;
                 Count:LongInt;
             Var Actual:LongInt)
```

Ecode:	Error indicator
Path:	Name of object containing the label
Data_addr:	Source or destination of I/O
Count:	Number of bytes to transfer
Actual:	Actual number of bytes transferred

These calls read or write the label of an object in the file system. I/O always starts at the beginning of the label. Count is the number of bytes the process wants transferred to or from Data_addr, and Actual is the actual number of bytes transferred. An error is returned if you attempt to read more than the maximum label size. A label can never be longer than 128 bytes, so you can never read or write more than that.

```
DEVICE_CONTROL (Var Ecode:Integer;
                Var Path:Pathname;
                Var CParm:dctype)

        Ecode: Error indicator
        Path:  Device to be controlled
        CParm: A record of information for the device driver
```

DEVICE_CONTROL is used to send device-specific information to a device driver, or to obtain device-specific information from a device driver.

Regardless of whether you are setting device control parameters or requesting information, you always use a record of type dctype. The structure of dctype is:

```
Dctype = RECORD
         dcVersion: INTEGER;
         dcCode:    INTEGER;
         dcData:    ARRAY[0..9] OF LONGINT
         END;

dcVersion: always 2 for the functions discussed in this
           document
dcCode:    control code for device driver
dcData:    specific control or data parameters
```

DEVICE_CONTROL functions that set attributes for a device driver are covered first.

CONTROLLING DEVICES

Before you use a device, you should use DEVICE _CONTROL in order to set the device driver so that it properly handles the device. Once you begin using the device, you are free to call DEVICE_CONTROL as necessary.

Following are two tables. The first, Table 2-1, shows which "groups" of device control functions must be set before using each type of device. The second table shows which type of characteristics are contained in the groups. For example, you must set Group A for RS-232 input. If you look in Table 2-2, you see that Group A indicates the type of parity used with the device. Note that each group requires a separate call to DEVICE_CONTROL, and that you can only set one characteristic from each group. If you set more than one from the same group for a particular device, the last one set will apply.

Table 2-1
DEVICE_CONTROL FUNCTIONS REQUIRED
BEFORE USING A DEVICE

Device Type	Device Name	Required Groups
Serial RS-232 for input	RS232A or RS232B	A, C, D, E, F, G
Serial RS-232 for output or printer	RS232A or RS232B	A, B, C, G, H, I
Profile	SLOTxCHANy (where x and y are numbers) or PARAPORT	J
Parallel printer	SLOTxCHANy (where x and y are numbers) or PARAPORT	I
Console screen and keyboard	MAINCONSOLE or ALTCONSOLE	I
Diskette drive	UPPER or LOWER	J

Here is a sample program which shows how a device control parameter is set. This program sets the parity attribute for the RS232B port to "no parity". Note that the parity attribute only requires that you set cparm.dccode and cparm.dcdata[0]. Other parameters require that you also set cparm.dcdata[1] and cparm.dcdata[2]. They are set in a similar manner.

```
VAR
   cparm: dctype;
   errnum: integer;
   path: pathname;

BEGIN
   path:='-RS232B';
   cparm.dcversion:=2;   (* always set this value *)
   cparm.dccode:=     1;
   cparm.dcdata[0]:=  0;
   DEVICE_CONTROL(errnum,path,cparm);
END;
```

Table 2-2 shows how to set cparm.dccode, cparm.dcdata[0], cparm.dcdata[1], and cparm.dcdata[2] for the various available attributes. Note that any values in cparm.dcdata past cparm.dcdata[2] are ignored when you are setting attributes documented here.

Table 2-2

DEVICE_CONTROL OUTPUT FUNCTIONAL GROUPS

FUNCTION	.dccode	.dcdata[0]	.dcdata[1]	.dcdata[2]
Group A--Parity:				
No parity	1	0	--	--
Odd parity, no input parity checking	1	1	--	--
Odd parity, input parity errors = 00	1	2	--	--
Even parity, no input parity checking	1	3	--	--
Even parity, input parity errors = $80	1	4	--	--
Group B--Output Handshake:				
None	11	--	--	--
DTR handshake	2	--	--	--
XON/XOFF handshake	3	--	--	--
delay after Cr, LF	4	ms delay	--	--
Group C--Baud rate:				
	5	baud	--	--
Group D--Input waiting:				
wait for full line	6	0	--	--
return whatever rec'd	6	1	--	--

Group E--Input handshake:

no handshake	7 9	0 -1	-- -1	-- 65
DTR handshake	7	--	--	--
XON/XOFF handshake	8	--	--	--

Group F--Input type-ahead buffer:

flush only	9	-1	-2	-2
flush & re-size	9	bytes	-2	-2
flush, re-size, and set threshold	9	bytes	low	hi

Group G--Disconnect Detection:

none	10	--	0	--
BREAK detected means disconnect	10	--	non-zero	--

Group H--Timeout on output (handshake interval):

no timeout	12	0	--	--
timeout enabled	12	seconds	--	--

Group I--Automatic linefeed insertion:

disable	17	0	--	--
enabled	17	1	--	--

Group J--Disk errors (set to 1 to enable, to 0 to disable):

enable sparing	21	sparing	rewrite	reread

Group K--Break command (never required -- available only on serial RS-232 devices)

send break	13	millisecond duration	0	--
send break while lowering DTR	13	millisecond duration	1	--

Using Group C, you can set baud to any standard rate. However, 3600, 7200, and 19200 baud are available only on the RS232B port.

'Low' and 'Hi' under Group F set the low and high threshold in the type ahead input buffer. When 'hi' or more bytes are in the input buffer XOFF is sent or DTR is dropped. Then when 'Low' or fewer bytes are in the type ahead buffer, XON is sent or DTR is re-asserted. The size of the type ahead buffer (bytes) can be any value between 0 and 64 bytes inclusive.

In Group J, disk sparing, when enabled, orders the device driver to re-locate blocks of data from areas of the disk that are found to be bad.

Disk rewrite, when enabled, orders the Operating System to rewrite data that it had trouble reading, but finally managed to read. This condition is referred to as a *soft error.*

Disk reread, when enabled, orders the Operating System to read data after they are written, to make certain that they were written correctly.

When sending a break command, shown in Group K, any device control from Group A removes the break condition, even if the allotted time has not yet elapsed. Also, sending a break will disrupt transmission of any other character still being sent. If you want to make certain that enough time has elapsed for the last character to be transmitted, call WRITE_DATA with a single null character (equal to 0) just prior to calling DEVICE_CONTROL to send the break.

Table 2-3 gives a list of mneumonic constants that you can use in place of explicit numbers when setting dccode. These mneumonics are provided solely for convenience.

Table 2-3
Dccode Mneumonics

Dccode	Mneumonic
1	dvParity
2	dvOutDTR
3	dvOutXON
4	dvOutDelay
5	dvBaud
6	dvInWait
7	dvInDTR
8	dvInXON
9	dvTypeahd
10	dvDiscon
11	dvOutNoHS
12	no mneumonic
13	no mneumonic
15	dvErrStat

```
16          dvGetEvent
17          dvAutoLF
20          dvDiskStat
21          dvDiskSpare
```

OBTAINING DEVICE CONTROL INFORMATION

When you use DEVICE_CONTROL to find out information about the current state of a particular device, you simply give the pathname for the particular device,along with a function code for the type of information you need, and the record of type dctype that you supply is returned filled with information.

There are three types of information requests you can make. Note that each type only applies to some of the available devices. The request types and the returned information are described in Table 2-4.

Here is a program fragment that gets information about the upper diskette drive.

```
VAR
   cparm: dctype;
   errnum: integer;
   path: pathname;
BEGIN
   path:='-UPPER';
   cparm.dcversion:=2;    (* always set this value *)
   cparm.dccode := 20;
   DEVICE_CONTROL(errnum,path,cparm);
   WRITELN (dcdata[0],dcdata[1],dcdata[2],dcdata[3],
            dcdata[4],dcdata[5],dcdata[6])
END;
```

Table 2-4
Device Information

DCCODE	DEVICES	DCDATA (returned)
15	profiles	[0] contains disk error status on last hardware error (Table 2-5)
		[1] contains error retry count since last system boot
16	console screen and keyboard	[0] contains numbers 0 - 10, which indicate events: 0 = no event 1 = upper diskette inserted 2 = upper diskette button 3 = lower diskette inserted 4 = lower diskette button 6 = mouse button down 7 = mouse plugged in 8 = power button 9 = mouse button up 10 = mouse unplugged [1] contains the current state of certain keys, indicated by set bits (if the bit is 1, the key is pressed) (bits are numbered from the right) 0 = caps lock key 1 = shift key 2 = option key 3 = command key 4 = mouse button 5 = auto repeat [2] contains X and Y coordinates of mouse, X in left 2 bytes, Y in right 2 bytes [3] contains timer value, in milliseconds

20	profile or diskette drive	[0] contains: 0 = no disk present 1 = disk present (but not accessed yet) The following indicate that a disk is present, and it has been accessed at least once. 2 = bad block track appears unformatted 3 = disk formatted by some program other than the Operating System 4 = OS formatted disk [1] contains: 0 = no button press pending 1 = button press pending, disk not yet ejected [2] contains number of blocks (0-16) (meaningful only when dcdata[0] = 4 and for a diskette) [3] contains: 0 = both copies of the bad-block directory OK 1 = one copy is corrupt (meaningful only when dcdata[0] = 4) [4] contains: 0 = sparing disabled 1 = sparing enabled [5] contains: 0 = rewrite disabled 1 = rewrite enabled [6] contains: 0 = reread disabled 1 = reread enabled

Table 2-5 shows the breakdown of the error code in response to a dccode=15 information request. This code is given in cparm.dcdata[0].

The code is a long integer, and therefore contains four bytes. Each bit in every byte but byte 0 has meaning. The bytes are shown in Table 2-5 broken

up into bits, with the bits and bytes numbered from the right, counting from 0, as shown in Figure 2-1. In all cases, the meaning attributed to the bit is true if the bit is set (equals 1).

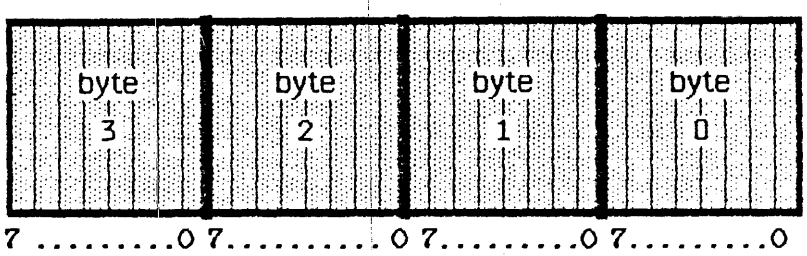

Figure 2-1
Disk Hard Error Codes

Table 2-5
Disk Hard Error Codes

Byte 3

7 = Profile received <> 55 to its last response
6 = Write or write/verify aborted because more than 532 bytes of data were sent or because profile could not read its spare table
5 = Hosts data is no longer in RAM because profile updated its spare table
4 = SEEK ERROR -- unable in 3 tries to read 3 consecutive headers on a track
3 = CRC error (only set during actual read or verify of write/verify, not while trying to read headers after seeking)
2 = TIMEOUT ERROR (could not find header in 9 revolutions) -- not set while trying to read headers after seeking
1 = Not used
0 = Operation unsuccessful

Byte 2

7 = SEEK ERROR -- unable in 1 try to read 3 consecutive headers on a track
6 = Spared sector table overflow (more than 32 sectors spared)
5 = Not used
4 = Bad block table overflow (more than 100 bad blocks in table)
3 = Profile unable to read its status sector
2 = Sparing occurred
1 = Seek to wrong track occurred
0 = Not used

Byte 1
- 7 = Profile has been reset
- 6 = Invalid block number
- 5 = Not used
- 4 = Not used
- 3 = Not used
- 2 = Not used
- 1 = Not used
- 0 = Not used

Byte 0

This byte contains the number of errors encountered when rereading a block after any read error.

```
ALLOCATE (Var Ecode:Integer;
          Refnum:Integer;
          Contiguous:Boolean;
          Count:Longint;
      Var Actual:Integer)
```

Ecode:	Error indicator
Refnum:	Reference number of object to be allocated space
Contiguous:	True=allocate contiguously
Count:	Number of blocks to be allocated
Actual:	Number of blocks actually allocated

Use **ALLOCATE** to increase the space allocated to an object. If possible, **ALLOCATE** adds the requested number of blocks to the space available to the object referenced by Refnum. The actual number of blocks allocated is returned in actual. If contiguous is true, the new space is allocated in a single, unfragmented space on the disk. This space is not necessarily adjacent to any existing file blocks.

ALLOCATE applies only to objects on block structured devices. An attempt to allocate more space to a pipe is successful only if the pipe's read pointer is less than or equal to its write pointer. If the write pointer has wrapped around, but the read pointer has not, an allocation would cause the reader to read invalid and uninitialized data, so the File System returns an error 1186 in this case.

```
COMPACT (Var Ecode:Integer;
         Refnum:Integer)
```

Ecode: Error indicator
Refnum: Reference number of object to be compacted

COMPACT deallocates any blocks after the block that contains the logical end of file for the file referenced by refnum. (See Figure 2-1.) COMPACT applies only to objects on block structured devices. As is the case with ALLOCATE, compaction of a pipe is legal only if the read pointer is less than or equal to the write pointer. If the write pointer has wrapped around, but the read pointer has not, compaction could destroy data in the pipe. The File System returns an error 1188 in this case.

```
TRUNCATE (Var Ecode:Integer;
          Refnum:Integer)

        Ecode:  Error indicator
        Refnum: Reference number of object to be truncated
```

TRUNCATE sets the logical end of file indicator to the current position of the file marker. Any data beyond the file marker are lost. TRUNCATE applies only to block structured devices. Truncation of a pipe can destroy data that have been written but not yet read. As the diagram shows, TRUNCATE does not change PEOF, only LEOF. COMPACT, on the other hand, changes both.

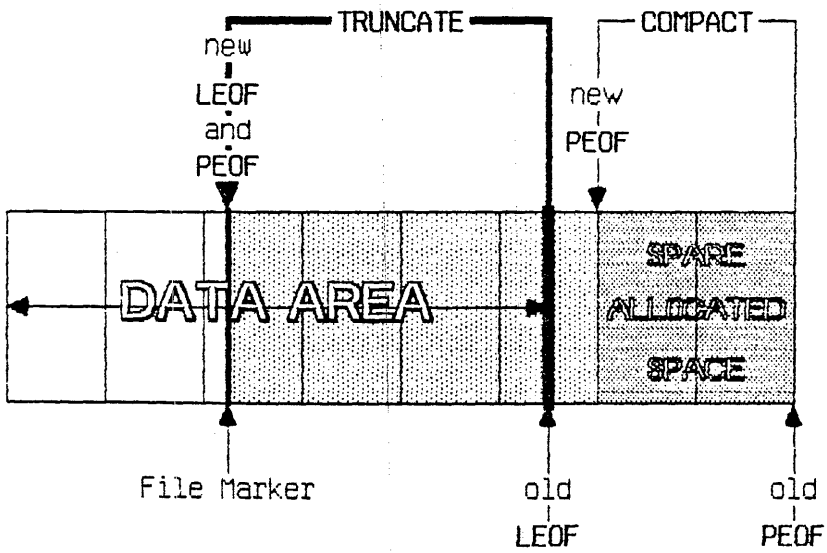

Figure 2-2
The Relationship of COMPACT and TRUNCATE

In this figure the boxes represent blocks of data. Note that LEOF can point to any byte in the file, but PEOF can only point to a block boundary. Therefore, TRUNCATE can reset LEOF to any byte in the file, but COMPACT can only reset PEOF to a block boundary.

```
FLUSH (Var Ecode:Integer;
       Refnum:Integer)

       Ecode:  Error indicator
       Refnum: Reference number of destination of I/O
```

FLUSH forces all buffered information destined for the object identified by refnum to be written out to that object.

A side effect of flush is that all FS buffers and data structures are flushed (as well as the control information for the referenced file). If Refnum is -1, only the global file system is flushed. This is a method by which an application can insure that the file system in consistent.

```
SET_SAFETY (Var Ecode:Integer;
            Var Path:Pathname;
               On_off:Boolean)
```

```
Ecode:    Error indicator
Path:     Name of object containing safety switch
On_Off:   Set saftey switch (On=true), or clear it
          (Off=false)
```

Each object in the file system has a "safety switch" to help prevent accidental deletion. If the safety switch is on, the object cannot be deleted. SET_SAFETY turns the switch on or off for the object identified by path. Processes that are sharing an object should cooperate with each other when setting or clearing the safety switch.

```
SET_WORKING_DIR (Var  Ecode:Integer;
                 Var Path:Pathname)

GET_WORKING_DIR (Var Ecode:Integer;
                 Var Path:Pathname)
```

```
Ecode:    Error indicator
Path:     Working directory name
```

The Operating System uses the name of the working directory to resolve partially specified pathnames into complete pathnames. GET_WORKING_DIR returns the current working directory name in Path. SET_WORKING_DIR sets the working directory name.

The following program fragment reports the current name of the working directory and allows you to set it to something else:

```
VAR  WorkingDir:PathName;
     ErrorCode:INTEGER;
BEGIN
  GET_WORKING_DIR(ErrorCode,WorkingDir);
  IF (ErrorCode<>0) THEN
    WRITELN('Cannot get the current working directory!')
  ELSE WRITELN('The current working directory is:
                ',WorkingDir);
  WRITE('New working directory name: ');
  READLN(WorkingDir);
  SET_WORKING_DIR(ErrorCode,WorkingDir);
END;
```

```
RESET_CATALOG (Var Ecode:INTEGER;
               Var Path:Pathname)

GET_NEXT_ENTRY (Var Ecode:INTEGER;
                Var Prefix,
                    Entry:E_Name)
```

```
Ecode:    Error indicator
Path:     Working directory name
Prefix:   Beginning of file names returned
Entry:    Names from catalog
```

RESET_CATALOG and GET_NEXT_ENTRY give a process access to catalogs. RESET_CATALOG sets the 'catalog file marker' to the beginning of the catalog specified by Path. Path should be a root volume name. GET_NEXT_ENTRY then performs sequential reads through the catalog file specified in the RESET_CATALOG call and returns file system object names. An end of file error code (848) is returned when GET_NEXT_ENTRY reaches the end of the catalog. If prefix is non-null, only those entries in the catalog that begin with that prefix are returned. If prefix is 'AB', for example, only file names that begin with 'AB' are returned. The prefix and catalog marker are local to the calling process, so several processes can simultaneously read a catalog without affecting each other.

```
MOUNT   (Var Ecode:Integer;
         Var VName:E_Name;
         Var Password:E_Name
         Var Devname:E_Name)

UNMOUNT (Var Ecode:Integer;
         Var Vname:E_name)
```

```
Ecode:    Error indicator
Vname:    Volume name
Password: Password for device (currently ignored)
Devname:  Device name
```

MOUNT and UNMOUNT handle access to sequential devices or block structured devices. For block structured devices, MOUNT logically attaches that volume's catalog to the file system. The name of the volume mounted is returned in the parameter Vname.

UNMOUNT detaches the specified volume from the file system. No object on that volume can be opened after UNMOUNT has been called. The volume cannot be unmounted until all the objects on the volume have been closed by all processes using them.

Devname is the name of the device on which a volume is being mounted. Devname should be given without a leading dash (-).

Vname is the name of the volume that was successfully mounted, and is returned.

Chapter 3
PROCESSES

3.1 Process Structure 3-4
3.2 Process Hierarchy 3-4
3.3 Process Creation 3-5
3.4 Process Control 3-5
3.5 Process Scheduling 3-6
3.6 Process Termination 3-6
3.7 A Process Handling Example 3-7
3.8 Process System Calls 3-9
MAKE_PROCESS 3-10
TERMINATE_PROCESS 3-11
INFO_PROCESS 3-13
KILL_PROCESS 3-15
SUSPEND_PROCESS 3-16
ACTIVATE_PROCESS 3-17
SETPRIORITY_PROCESS 3-18
YIELD_CPU 3-19
MY_ID 3-20

PROCESSES

A *process* is the entity in the Lisa system that performs work. When you ask the Operating System to run a program, the OS creates a specific instance of the program and its associated data. That instance is a process.

The Lisa can have a number of processes at any one time, and they will appear to be running simultaneously. Although processes can share code and data, each process has its own stack.

Actually, only one process can use the CPU at a time. Which process is active at a particular time is determined by the *scheduler.* The scheduler allows each process to run until some condition that would slow execution occurs (an I/O request, for example). At that time, the running process is saved in its current state, and the scheduler checks the pool of ready-to-run processes. When the original process later resumes execution, it picks up where it left off.

The process scheduling state has three possibilities. The process is *running* if it is actually engaging the attention of the CPU. If it is ready to execute, but is being held back by the scheduler, the process is *ready.* A process can also be *blocked.* In the blocked state, the process is ignored by the scheduler. It cannot continue its execution until something causes its state to be changed to ready. Processes commonly become blocked while awaiting completion of I/O, although there are a number of other likely causes.

3.1 Process Structure

A process can use up to 16 data segments and 106 code segments.

The layout of the process address space for user processes is shown in Figure 3-1.

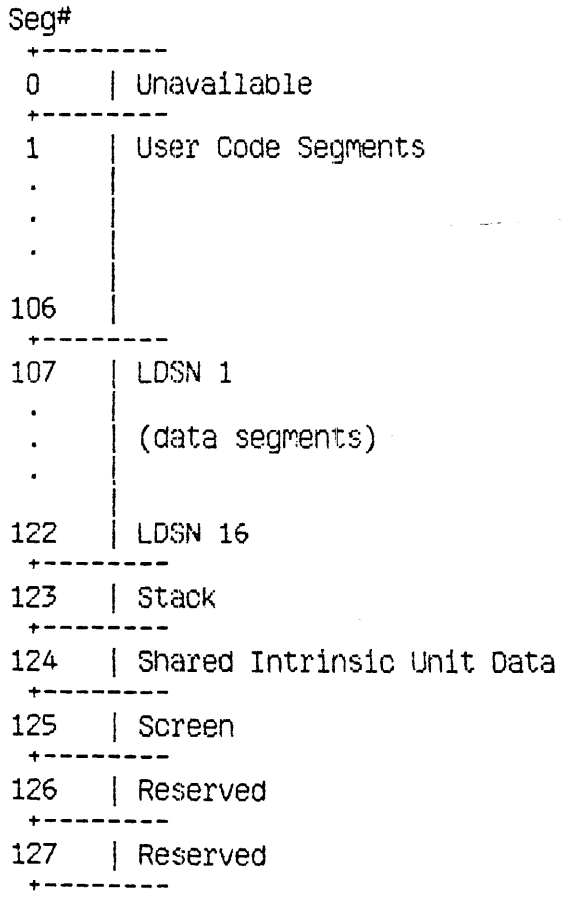

Figure 3-1
PROCESS ADDRESS SPACE LAYOUT

Each process has an associated priority, an integer between 1 and 255. The process scheduler usually executes the highest priority ready process. The higher priorities (226 to 255) are reserved for Operating System.

3.2 Process Hierarchy

When the system is first started, several system processes exist. At the base of the process hierarchy is the root process which handles various internal Operating System functions. It has at least three sons, the memory manager process, the timer process, and the shell process.

The memory manager process handles code and data segment swapping.

The shell process is a user process which is automatically started when the OS is initialized. It typically is a command interpreter, but it can be any program. The OS simply looks for the program called $YSTEM.SHELL, and executes it.

The timer process handles timing functions such as timed event channels.

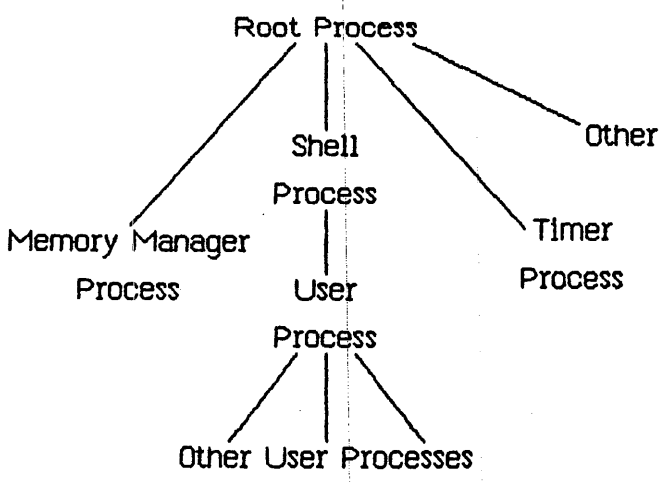

Figure 3-2
Process Tree

Any other system process (the Network Control Process, for example) is a son of the root process.

3.3 Process Creation

When a process is created, it is placed in the ready state, with a priority equal to that of the process which created it. All the processes created by a given process can be thought of as existing in a subtree. Many of the process management calls can affect the entire subtree of a process as well as the process itself.

3.4 Process Control

Three system calls are provided for explicit control of a process. These calls allow a process to kill, suspend (block), or activate any other user process in the system, as long as the process identifier is known. Process handling calls are not allowed on Operating System processes.

3.5 Process Scheduling

Process scheduling is based on the priority established for the process and on requests for Operating System services.

The scheduler generally executes the highest priority ready process. Once a process is executing, it loses the CPU only under certain circumstances. In practice, the CPU is lost when there is some specific request for the process to wait (for an event, for example), when there is an I/O request, or when there is a reference to a code segment that is not in memory. A process that makes any Operating System call may lose the CPU. The process will get the CPU back when the Operating System is finished except under the following conditions:

- The running process requests input or output. The scheduler will start the next highest-priority process running while the first process waits for the I/O to complete.
- The running process lowers its priority below that of another ready process or sets another process's priority to be higher than its own.
- The running process explicitly yields the CPU to another process.
- The running process activates a higher priority process.
- The running process suspends itself.
- A higher priortity process becomes ready.
- The running process needs code to be swapped into memory.
- The running process executes an event wait call.
- The running process calls DELAY_TIME.

Because the Operating System cannot seize the CPU from an executing process except in the cases noted above, background processes should be liberally sprinkled with YIELD_CPU calls.

When the scheduler is invoked, it saves the state of the current process and selects the next process to run by examining the pool of ready processes. If the new process requires that code or data be loaded into memory, the memory manager process is launched. If the memory manager is already working on a process, the scheduler selects the highest priority process in the ready queue that does not need anything swapped.

3.6 Process Termination

A process terminates normally when it calls TERMINATE_PROCESS, when it reaches an 'END.' statement, when some process calls KILL_PROCESS on it, when its father process terminates, or when it runs into an abnormal condition. When a process begins to terminate, a SYS_TERMINATE exception condition is signalled to the terminating process and all of the processes it has created. Any

process can create an exception handler with the DECLARE_EXCEP_HDL call (described in Chapter 5 of this manual), so the process can catch the terminate exception and clean up before terminating. The SYS_TERMINATE exception handler will only be executed once. Thus, if an error occurs while the handler is executing, the process terminates immediately.

Termination involves the following steps:

1. Signal the SYS_TERMINATE exception on the terminating process.
2. Execute the user's exception handler (if any).
3. Instruct all sons of the current process to terminate.
4. Close all open files, data segments, pipes, and event channels left open by the user process.
5. Send the SYS_SON_TERM event to the father of the terminating process if a local event channel exists.
6. Wait for all the sons to finish termination.

3.7 A Process Handling Example

The following programs illustrate the use of many of the process management calls described in this chapter. The program FATHER creates a son process, and lets it run for awhile. It then gives you a chance to activate, suspend, kill, or get information about the son.

```
PROGRAM Father;
USES (*$U Source:SysCall.Obj*) SysCall;
VAR ErrorCode:INTEGER;(*error returns from system calls *)
    proc_id:LONGINT;        (* process global identifier *)
    progname:Pathname;        (* program file to execute *)
    null:NameString;             (* program entry point *)
    Info_Rec:ProcInfoRec;  (* information about process *)
    i:INTEGER;
    Answer:CHAR;
BEGIN
   ProgName:='SON.OBJ'; (* this program is defined below *)
   Null:='';
   MAKE_PROCESS(ErrorCode,Proc_Id,ProgName,Null,0);
   IF (ErrorCode<>0) THEN
      WRITELN('Error ',ErrorCode,' during process management.');
   FOR i:=1 TO 15 DO                (* idle for awhile *)
   BEGIN
      WRITELN('Father executes for a moment.');
      YIELD_CPU(ErrorCode,FALSE);   (* let son run *)
   END;
   WRITE('K(ill S(uspend A(ctivate I(nfo');
```

```
        READLN(Answer);
        CASE Answer OF
           'K','k': KILL_PROCESS(ErrorCode,Proc_Id);
           'S','s': SUSPEND_PROCESS(ErrorCode,Proc_Id,TRUE
                                    (* suspend family *));
           'A','a': ACTIVATE_PROCESS(ErrorCode,Proc_Id,TRUE
                                    (* activate family *));
           'I','i': BEGIN
           INFO_PROCESS(ErrorCode,Proc_Id,Info_Rec);
           WRITELN('Son''s name is ',Info_Rec.ProgPathName);
           END;
        END;
        IF (ErrorCode<>0) THEN
           WRITELN('Error ',ErrorCode,' during process
                    management.');
END.
```

The program SON is:

```
PROGRAM Son;
USES (*$U Source:SysCall.Obj*) SysCall;
VAR ErrorCode:INTEGER;
    null:NameString;
BEGIN
   WHILE TRUE DO
      BEGIN
         WRITELN('Son executes for a moment.');
         YIELD_CPU(ErrorCode,FALSE);(*let father process run*)
      END;
END.
```

3.8 Process System Calls

This section describes all the Operating System calls that pertain to process control. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in process control calls:

```
Pathname = STRING[255];
Namestring = STRING[20];
P_s_eventblock = ^s_eventblock;
S_eventblock = T_event_text;
T_event_text = array [0..size_etext] of longint;
ProcinfoRec = record
                progpathname : pathname;
                global_id    : longint;
                father_id    : longint;
                priority     : 1..255;
                state   : (pactive, psuspended, pwaiting);
                data_in : boolean
                end;
```

```
MAKE_PROCESS (Var ErrNum:Integer;
              Var Proc_id:LongInt;
              Var ProgFile:Pathname;
              Var EntryName:NameString; (* NameString = STRING[20] *)
                  Evnt_chn_refnum:Integer)
```

ErrNum:	Error indicator
Proc_id:	Process identifier (globally unique)
ProgFile:	Process file name
EntryName:	Program entry point
Evnt_chn_refnum:	Communication channel between calling process and created process

A process is created when another process calls MAKE_PROCESS. The new process executes the program identified by the pathname, progfile. If progfile is a null character string, the program name of the calling process is used. A globally unique identifier for the created process is returned in proc_id.

Evnt_chn_refnum is a local event channel supplied by the calling process (event channels are discussed in Chapter 5 of this manual). The Operating System uses the event channel identified by evnt_chn_refnum to send the calling process events regarding the created process (for example, SYS_SON_TERM). If evnt_chn_refnum is zero, the calling process is not informed when such events are produced.

Entryname, if non-null, specifies the program entry point where execution is to begin. Because alternate entry points have not yet been defined, this parameter is currently unused.

Any error encountered during process creation is reported in ErrNum.

```
TERMINATE_PROCESS(Var ErrNum:Integer;
                  Event_ptr:P_s_eventblk)

        ErrNum:    Error indicator
        Event_ptr: Information sent to process's creator
```

The life of a process can be ended by TERMINATE_PROCESS. This call causes a SYS_TERMINATE exception to be signalled on the calling process and on all of the processes it has created. The process can declare its own SYS_TERMINATE exception handler to handle whatever cleanup it needs to do before it is actually terminated by the system. When the terminate exception handler is entered, the exception information block contains a longint that describes the cause of the process termination:

Excep_Data[0] =
- 0 Process called TERMINATE_PROCESS
- 1 Process executed the 'END.' statement
- 2 Process called KILL_PROCESS on itself
- 3 Some other process called KILL_PROCESS on the terminating process
- 4 Father process is terminating
- 5 Process made an invalid system call (that is, an unknown call)
- 6 Process made a system call with an invalid errnun parameter address
- 7 Process aborted due to an error while trying to swap in a code or data segment
- 8 Process exceeded its maximum specified stack size
- 9 Process aborted due to possible lock up of the system by a data space exceeding physical memory size
- 10 Process aborted due to a parity error

There are an additional twenty-six errors that can be signaled. The entire list is shown on the first page of Appendix A.

If the terminating process was created with a communication channel, a SYS_SON_TERM event is sent to the terminating process's father. The terminating process can specify the text of the SYS_SON_TERM with the Event_ptr parameter. Note that the first (0'th) longint of the event text is reserved by the system. When the event is sent to the father, the OS places the

termination cause of the son process in the first longint. This is the same termination cause that was supplied to the terminating process itself in the SYS_TERMINATE exception information block. Any user-supplied data in the first longint of the event text is overwritten.

If a process specifies an event to be sent in the TERMINATE_PROCESS call, but the process was created without a local event channel, no event is sent to the father.

If a process terminates by a means other than calling TERMINATE_PROCESS, or it specifies a nil Event_ptr in the TERMINATE_PROCESS call, and the process was created with a local event channel, an event is sent to the father that contains the termination cause in the first longint and zeros in the remaining event text.

P_s_eventblk is a pointer to an s_eventblk. S_eventblk is defined as:

```
CONST size_etext = 9;  (* event text size - 40 bytes *)
TYPE t_event_text = ARRAY [0..size_etext] OF LongInt;
     s_eventblk   = t_event_text;
```

If a process calls TERMINATE_PROCESS twice, the Operating System forces it to terminate even if it has disabled the terminate exception.

```
INFO_PROCESS (Var ErrNum:Integer;
                  Proc_Id:LongInt;
              Var Proc_Info:ProcInfoRec);
```

ErrNum:	Error indicator
Proc_Id:	Global identifier of process
Proc_Info:	Information about the process identified by Proc_id

A process can call INFO_PROCESS to get a variety of information about any process known to the Operating System. Use the function MY_ID to get the Proc_Id of the calling process.

ProcInfoRec is defined as:

```
TYPE ProcInfoRec = RECORD
  ProgPathname:Pathname;
  Global_id    :Longint;
  Priority     :1..255;
  State        :(PActive,PSuspended,PWaiting);
  Data_in      :Boolean
END;
```

Data_In indicates whether the data space of the process is currently in memory.

The following procedure gets information about a process and displays some of it:

```
PROCEDURE Display_Info(Proc_Id:LONGINT);
VAR ErrorCode:INTEGER;
    Info_Rec:ProcInfoRec;
BEGIN
   INFO_PROCESS(ErrorCode,Proc_Id,Info_Rec);
   IF (ErrorCode=100) THEN
      WRITELN('Attempt to display info about nonexistent
               process.')
   ELSE
   BEGIN
      WITH Info_Rec DO
      BEGIN
         WRITELN(' program name: ',ProgPathName);
         WRITELN(' global id:    ',Global_id);
         WRITELN(' priority:     ',priority);
         WRITE(' state:          ');
```

```
            CASE State OF
               PActive:    WRITELN('active');
               PSuspended: WRITELN('suspended');
               PWaiting:   WRITELN('waiting')
            END
         END
      END
   END;
```

```
KILL_PROCESS (Var ErrNum:Integer;
              Proc_Id:LongInt)

        ErrNum:      Error indicator
        Proc_Id:     Process to be killed
```

KILL_PROCESS kills the process referred to by proc_id and all of the processes in its subtree. The actual termination of the process does not occur until the process is in one of the following states:

- Executing in user mode.
- Stopped due to a SUSPEND_PROCESS call.
- Stopped due to a DELAY_TIME call.
- Stopped due to a WAIT_EVENT_CHN or SEND_EVENT_CHN call, or READ_DATA or WRITE_DATA to a pipe.

```
SUSPEND_PROCESS (Var ErrNum:Integer;
                     Proc_id:LongInt;
                     Susp_Family:Boolean)

        ErrNum:      Error indicators
        Proc_Id:     Process to be suspended
        Susp_Family: If true, suspend the entire process subtree
```

SUSPEND_PROCESS allows a process to suspend (block) any process in the system. The actual suspension does not occur until the process referred to by proc_id is in one of the following states:

- o Executing in user mode
- o Stopped due to a DELAY_TIME call
- o Stopped due to a WAIT_EVENT_CHN call

Neither expiration of the delay time nor receipt of the awaited event causes a suspended process to resume execution. SUSPEND_PROCESS is the only direct way to block a process. Processes, however, can become blocked during I/O, and by the timer (see DELAY_TIME), and for many other reasons.

If susp_family is true, the Operating System suspends both the process referred to by proc_id and all of its descendents. If susp_family is false, only the process identified by proc_id is suspended.

```
ACTIVATE_PROCESS(Var ErrNum:Integer;
                     Proc_Id:LongInt;
                     Act_Family:Boolean)
```

```
ErrNum:      Error indicator
Proc_Id:     Process to be activated
Act_Family:  If true, activate the entire process
subtree
```

To awaken a suspended process, call ACTIVATE_PROCESS. A process can activate any other process in the system. Note that ACTIVATE_PROCESS can only awaken a suspended process. If the process is blocked for some other reason, ACTIVATE_PROCESS cannot unblock it. If act_family is true, ACTIVATE_PROCESS also activates all the descendents of the process referred to by proc_id.

```
SETPRIORITY_PROCESS(Var ErrNum:Integer;
                    Proc_Id:LongInt;
                    New_Priority:Integer)

        ErrNum:       Error indicator
        Proc_id:      Global id of process
        New_Priority: Process's new priority number
```

SETPRIORITY_PROCESS changes the scheduling priority of the process referred to by proc_id to new_priority. The higher the priority value (which must be between 1 and 225), the more likely the process is to be allowed to execute. (Operating System processes execute with priorities between 226 and 255.)

```
YIELD_CPU(Var ErrNum:Integer;
          To_Any:Boolean)

        ErrNum: Error indication
        To_Any: Yield to any process, or only higher or equal
                priority
```

If To_Any is false, YIELD_CPU causes the calling process to give the CPU to any other ready-to-execute process with an equal or higher priority. If To_Any is true, YIELD_CPU causes the calling process to yield the CPU to any other ready process. If no such process exists, the calling process simply continues execution. Successive yields by processes of the same priority result in a "round-robin" scheduling of the processes. Background processes should use YIELD_CPU often to allow other processes to execute when they need to.

`MY_ID:Longint`

MY_ID is a function that returns the unique global identifier (a longint) of the calling process. A process can use MY_ID to perform process handling calls on itself.

For example:

```
SetPriority_Process(Errnum, My_Id, 100)
```

sets the priority of the calling process to 100.

Chapter 4
MEMORY MANAGEMENT

4.1 Data Segments 4-3
4.2 The Logical Data Segment Number 4-3
4.3 Shared Data Segments 4-4
4.4 Private Data Segments 4-4
4.5 Code Segments 4-4
4.6 Swapping 4-5
4.7 Memory Management System Calls 4-5
MAKE_DATASEG 4-6
KILL_DATASEG 4-8
OPEN_DATASEG 4-9
CLOSE_DATASEG 4-10
FLUSH_DATASEG 4-11
SIZE_DATASEG 4-12
INFO_DATASEG 4-13
INFO_LDSN 4-14
INFO_ADDRESS 4-15
MEM_INFO 4-16
SETACCESS_DATASEG 4-17
BIND_DATASEG 4-18
UNBIND_DATASEG 4-18

MEMORY MANAGEMENT

Every process has a set of code and data segments which must be in physical memory when those code segments and data segments are used. The translation of the logical address used by the process to the physical address used by the memory controller to access physical memory is handled by the memory management unit (MMU).

4.1 Data Segments

Each process has a data segment that the Operating System automatically allocates to it for use as a stack. The stack segment's internal structures are managed directly by the hardware and the Operating System.

A process can acquire additional data segments for uses such as heaps and inter-process communication. These additional data segments can be *private* (or *local*) data segments or *shared* data segments. Private data segments can be accessed only by the creating process. When the process terminates, any private data segments still in existence are destroyed. Shared data segments can be accessed by any process that opens those segments. A shared data segment is permanent until explicitly killed by a process.

The Operating System requires that data segments be in physical memory before the data are referenced. The scheduler automatically loads all of the data segments which the program says it needs. It is the responsibility of the programmer to insure that the program declares all its needs by associating itself with the needed data segments before they are needed.

This process of association is called *binding*. A program can bind a data segment to itself in several ways. When a program creates a data segment by using the MAKE_DATASEG call, the segment is automatically opened and bound to the program. If a program needs to open a segment that was already created by another program, the OPEN_DATASEG call is used. That call binds the segment to the calling process, as well as opening the segment for the process. Since there may be times when a process needs to use more data segments than can be bound at one time, the UNBIND_DATASEG call is provided, which leaves the data segment open, but unbinds it. The program can then use BIND_DATASEG to bind another data segment to the program.

The Operating System views all data segments except the stack as linear arrays of bytes. Therefore, allocation, access, and interpretation of structures within a data segment are the responsibility of the program.

4.2 The Logical Data Segment Number

The address space of a process allows up to 16 data segments bound to a process at any instant, in addition to the stack. Each bound data segment is associated with a specific region of the address space with a Logical Data Segment Number

(LDSN). (See Figure 3-1.) While a data segment is bound to the process it is said to be a member of the working set of the process.

The process associates a data segmennt with a specific LDSN in the MAKE_DATASEG or OPEN_DATASEG call.

The LDSN, which has a valid range of 1 to 16, is local to the calling process. The process uses the LDSN to keep track of where a given data segment can be found. More than one data segment can be associated with the same LDSN, but only one such segment can be bound to an LDSN at any instant and thus be a member of the working set of the process.

4.3 Shared Data Segments

Cooperating processes can share data segments. Shared segments cannot be larger than 128 Kbytes in length. As with local data segments, the segment creator assigns the segment a file system pathname. All processes that want to share that data segment then use the same pathname. If the shared data segment contains address pointers to data within the segment, the cooperating processes must also use the same LDSN with the segment. This insures that all logical data addresses referencing locations within the data segment are consistent for the processes sharing the segment.

4.4 Private Data Segments

Data segments can also be private to a process. In this case, the maximum size of the segment can be greater than 128 Kbytes. The actual maximum size depends on the amount of physical memory in the machine and the number of adjacent LDSN's available to map the segment. The process gives the desired segment size and the base LDSN to use to map the segment. The Memory Manager then uses ascending adjacent LDSN's to map successive 128 Kbyte chunks of the segment. The process must insure that enough consecutive LDSN's are available to map the entire segment.

Suppose a process has a data segment already bound to LDSN 2. If the program tries to bind a 256 Kbyte data segment to LDSN 1, the Operating System returns an error because the 256 Kbyte segment needs two consecutive free LDSN's. Instead, the program should bind the segment to LDSN 3 and the system automatically also uses LDSN 4.

4.5 Code Segments

Division of a program into multiple code segments (swapping units) is dictated by the programmer through commands to the compiler and linker. The MMU registers can map up to 106 code segments.

4.6 Swapping

When a process executes, the following segments must be in physical memory:

- o The current code segment
- o All the data segments in the process working set (the stack and all bound data segments)

The Operating System insures that this minimum set of segments is in physical memory before the process is allowed to execute. If the program calls a procedure in a segment not in memory, a segment swap-in request is initiated. In the simplest case, this request only requires the system to allocate a block of physical memory and to read in the segment from the disk. In a worse case, the request may require that other segments be swapped out first to free up sufficient memory. A clock algorithm is used to determine which segments to swap out or replace. This process is invisible to the program.

4.7 Memory Management Calls

This section describes all the Operating System calls that pertain to memory management. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in memory management calls:

```
Pathname = STRING[255];
Tdstype = (ds_shared, ds_private);
DsinfoRec = Record
                mem_size:longint;
                disc_size:longint;
                numb_open:integer;
                LDSN:integer;
                boundF:boolean;
                presentF:boolean;
                creatorF:boolean;
                rwaccess:boolean;
                segptr:longint;
                volname:e_name;
            end;
E_name = string [32];
```

```
MAKE_DATASEG (Var ErrNum:Integer;
              Var Segname:Pathname;
                  Mem_Size, Disk_Size:LongInt;
              Var RefNum:Integer;
              Var SegPtr:LongInt;
                  Ldsn:Integer
                   Dstype:Tdstype)
```

ErrNum:	Error indicator
Segname:	Pathname of data segment
Mem_Size:	Bytes of memory to be allocated to data segment
Disk_Size:	Bytes on disk to be allocated for swapping segment
RefNum:	Identifier for data segment
SegPtr	Address of data segment
Ldsn:	Logical data segment number
Dstype:	Type of dataseg (shared or private)

MAKE_DATASEG creates the data segment identified by the pathname, segname, and opens it for immediate read-write access. Segname is a file system pathname.

The parameter Mem_size determines how many bytes of main memory the segment is allocated. The actual allocation takes place in terms of 512 byte pages. If the data segment is private (Dstype is ds_private), Mem_size can be greater than 128 Kbytes, but you must insure that enough consecutive LDSN's are free to map the entire segment.

Disk_size determines the number of bytes of swapping space to be allocated to the segment on disk. If Disk_size is less than Mem_size, the segment cannot be swapped out of main memory. In this case the segment is memory resident until it is killed or until its size in memory becomes less than or equal to its disk_size (see SIZE_DATASEG). The application programmer should be aware of the serious performance implications of forcing a segment to be memory resident. Because the segment cannot be swapped out, a new process may not be able to get all of its working set into memory. To avoid thrashing, each application should insure that all of its data segments are swappable before it relinquishes the attention of the processor.

The calling process associates a Logical Data Segment Number (LDSN) with the data segment. If this LDSN is bound to another data segment at the time of the call, the call returns an error.

Refnum is returned by the system to be used in any further references to the data segment. The Operating System also returns segptr, an address pointer to be used

to reference the contents of the segment. Segptr points to the base of the data segment.

Any error conditions are returned in ErrNum.

When a data segment is made, it immediately becomes a member of the working set of the calling process. You can use UNBIND_DATASEG to free the LDSN.

```
KILL_DATASEG (Var ErrNum:Integer;
              Var Segname:Pathname)

        ErrNum:   Error indicator
        Segname:  Name of data segment to be deleted
```

When a process is finished with a shared data segment, it can issue a KILL_DATASEG call for that segment. (KILL_DATASEG cannot be used on a private data segment.) If any process, including the calling process, still has the data segment open, the actual deallocation of the segment is delayed until all processes have closed it (see CLOSE_DATASEG). During the interim period, however, after a KILL_DATASEG call has been issued but before the segment is actually deallocated, no other process can open that segment.

KILL_DATASEG does not affect the membership of the data segment in the working set of the process. The refnum and segptr values are valid until a CLOSE_DATASEG call is issued.

One important note: Normally, when a data segment is closed, the contents are written to disk as a file with the pathname associated with the data segment. If, however, the program calls KILL_DATASEG on the data segment before closing the data segment, the contents of the data segment are not written to disk, and will be lost when the segment is closed.

```
OPEN_DATASEG (Var ErrNum:Integer;
              Var Segname:Pathname;
              Var RefNum:Integer;
              Var SegPtr:LongInt;
                  Ldsn:Integer)
```

ErrNum:	Error indicator
Segname:	Name of data segment to be opened
RefNum:	Identifier for data segment
SegPtr	Pointer to contents of data segment
Ldsn:	Logical data segment number

A process can open an existing shared data segment with OPEN_DATASEG. The calling process must supply the name of the data segment (segname) and the logical data segment number to be associated with it. The logical data segment number given must not have a data segment currently bound to it. The segment's name is determined by the process which creates the data segment; it cannot be null.

The Operating System returns both refnum, an identifier for the calling process to use in future references to the data segment, and segptr, an address pointer used to reference the contents of the segment.

When a data segment is opened, it immediately becomes a member of the working set of the calling process. The access mode of the newly opened segment is Readonly. You can use SETACCESS_DATASEG to change the access rights to Readwrite. You can use UNBIND_DATASEG to free the LDSN.

You cannot use OPEN on a private data segment, since calling CLOSE on a private data segment deletes it.

```
CLOSE_DATASEG (Var ErrNum:Integer;
               Refnum:Integer)

      ErrNum: Error indicator
      Refnum: Data segment identifier
```

CLOSE_DATASEG terminates any use of refnum for data segment operations. If the data segment is bound to a Logical Data Segment Number, CLOSE_DATASEG frees that LDSN. The data segment is removed from the working set of the calling process. Refnum is made invalid. Any references to the data segment using the original segptr will have unpredictable results.

If Refnum refers to a private data segment, CLOSE_DATASEG also kills the data segment, deallocating the memory and disk space used for the data segment. If refnum refers to a shared data segment, the contents of the data segment are written to disk as if FLUSH_DATASEG had been called. (If KILL_DATASEG is called before CLOSE_DATASEG, the contents of the data segment are thrown away when the last process closes the data segment.)

The following procedure sets up a heap for LisaGraf using the memory management calls:

```
PROCEDURE InitDataSegForLisaGraf (var ErrorCode:integer);
CONST HeapSize=16384;  (* 16 KBytes for graphics heap *)
   DiskSize=16384;
VAR HeapBuf:LONGINT;  (* pointer to heap for LisaGraf *)
   GrafHeap:PathName;        (* data segment path name *)
   Heap_Refnum:INTEGER;    (* refnum for heap data seg *)

BEGIN
   GrafHeap:='grafheap';
   OPEN_DATASEG(ErrorCode,GrafHeap,Heap_Refnum,HeapBuf,1);
   IF (ErrorCode<>0) THEN
   BEGIN
      WRITELN('Unable to open',Grafheap,'Error is ',
              ErrorCode)
   END
   ELSE
      InitHeap(POINTER(HeapBuf),POINTER(HeapBuf+HeapSize),
              @HeapError);
END;
```

```
FLUSH_DATASEG (Var ErrNum;
               Refnum:Integer)

        ErrNum: Error indicator
        Refnum: Data segment identifier
```

FLUSH_DATASEG writes the contents of the data segment identified by refnum to the disk. (Note that CLOSE_DATASEG automatically flushes the data segment before closing it, unless KILL_DATASEG was called first.) This call has no effect upon the memory residence or binding of the data segment.

```
SIZE_DATASEG (Var ErrNum:Integer;
                  Refnum:Integer;
                  DeltaMemSize:LongInt;
              Var NewMemSize:LongInt;
                  DeltaDiskSize:LongInt;
              Var NewDiskSize:LongInt)
```

ErrNum:	Error indicator
Refnum:	Data segment identifier
DeltaMemSize:	Amount in bytes of change in memory allocation
NewMemSize:	New actual size of segment in memory
DeltaDiskSize:	Amount in bytes of change in disk allocation
NewDiskSize:	New actual disk (swapping) allocation

SIZE_DATASEG changes the memory and/or disk space allocations of the data segment referred to by refNum. Both deltaMemSize and deltaDiskSize can be either positive, negative, or zero. The changes to the data segment take place at the high end of the segment and do not destroy the contents of the segment, unless data are lost in shrinking the segment. Because the actual allocation is done in terms of pages (512 byte blocks), the newMemSize and newDiskSize returned by SIZE_DATASEG may be larger than the oldsize plus deltaSize of the respective areas.

If the newDiskSize is less than the newMemSize, the segment cannot be swapped out of memory. The application programmer should be aware of the serious performance implications of forcing a segment to be memory resident. Because the segment cannot be swapped out, a new process may not be able to get all of its working set into memory. To avoid thrashing, each application should insure that all of its data segments are swappable before it relinquishes the attention of the processor.

If the necessary adjacent LDSN's are available, SIZE_DATASEG can increase the size of a private data segment beyond 128 Kbytes.

```
INFO_DATASEG (Var ErrNum:Integer;
                  Refnum:Integer;
              Var DsInfo:DsInfoRec)
```

ErrNum: Error indicator
Refnum: Identifier of data segment
DsInfo: Attributes of data segment

INFO_DATASEG returns information about a data segment to the calling process. The structure of the dsinforec record is:

```
RECORD
Mem_Size:LongInt    (*Bytes of memory allocated to data segment          *);
Disc_Size:LongInt   (*Bytes of disk space allocated to segment          *);
NumbOpen:Integer    (*Current number of processes with segment open     *);
Ldsn:Integer        (*LDSN for segment binding                          *);
BoundF:Boolean      (*True if segment is bound to LDSN of calling proc  *);
PresentF:Boolean    (*True if segment is present in memory              *);
CreatorF:Boolean    (*True if the calling process is the creator        *)
                    (*of the segment                                    *);
RWAccess:Boolean    (*True if the calling process has Write access      *)
                    (*to segment                                        *)
END;
```

```
INFO_LDSN ( Var ErrNum:Integer;
              Ldsn:Integer;
            Var RefNum:Integer)

        ErrNum: Error indicator
        Ldsn:   Logical data segment number
        RefNum: Data segment identifier
```

INFO_LDSN returns the refnum of the data segment currently bound to Ldsn. You can then use **INFO_DATASEG** to get information about that data segment. If the LDSN specified is not currently bound to a data segment, the refnum returned is -1.

```
INFO_ADDRESS (Var ErrNum:Integer;
                  Address:Longint;
              Var RefNum:Integer)
```

ErrNum: Error indicator
Address: The address about which the program needs information
RefNum: Data segment identifier

This call returns the refnum of the currently bound data segment that contains the address given.

If no data segment is currently bound to the calling process that contains the address given, an error indication is returned in ErrNum.

```
MEM_INFO (Var ErrNum:Integer;
          Var Swapspace;
              Dataspace;
              Cur_codesize;
              Max_codesize:Longint)
```

ErrNum:	Error indicator
Swapspace:	Amount, in bytes, of swappable system memory available to the calling process
Dataspace:	Amount, in bytes, of system memory that the calling process needs for its bound data areas, including the process stack and the shared intrinsic data segment
Cur_codesize:	Size, in bytes, of the calling segment
Max_codesize:	Size, in bytes, of the largest code segment within the address space of the calling process

This call retrieves information about the memory resources used by the calling process.

```
SETACCESS_DATASEG (Var ErrNum:Integer;
                       Refnum:Integer;
                       Readonly:Boolean)
```

```
ErrNum:    Error indicator
Refnum:    Data segment identifier
Readonly:  Access mode
```

A process can control the kinds of access it is allowed to exercise on a data segment with the SETACCESS_DATASEG call. Refnum is the identifier for the data segment. If readonly is true, an attempt by the process to write to the data segment results in an address error exception condition. To get readwrite access, set readonly to false.

```
BIND_DATASEG(Var ErrNum:Integer;
             RefNum:Integer)

UNBIND_DATASEG(Var ErrNum:Integer;
               RefNum:Integer)

    ErrNum: Error indicator
    RefNum: Data segment identifier
```

BIND_DATASEG binds the data segment referred to by refnum to its associated logical data segment number(s). UNBIND_DATASEG unbinds the data segment from its LDSN(s). BIND_DATASEG causes the data segment to become a member of the current working set. At the time of the BIND_DATASEG call, the necessary LDSN(s) must not be bound to a different data segment. UNBIND_DATASEG frees the associated LDSN(s). A reference to the contents of an unbound segment gives unpredictable results. OPEN_DATASEG and MAKE_DATASEG define which LDSN(s) is associated with a given data segment.

Chapter 5
EXCEPTIONS AND EVENTS

5.1 Exceptions 5-3
5.2 System Defined Exceptions 5-4
5.3 Exception Handlers 5-4
5.4 Events 5-7
5.5 Event Channels 5-7
5.6 The System Clock 5-12
5.7 Exception Management System Calls 5-12
DECLARE_EXCEP_HDL 5-13
DISABLE_EXCEP 5-14
ENABLE_EXCEP 5-15
INFO_EXCEP 5-16
SIGNAL_EXCEP 5-17
FLUSH_EXCEP 5-18
5.8 Event Management System Calls 5-18
MAKE_EVENT_CHN 5-20
KILL_EVENT_CHN 5-21
OPEN_EVENT_CHN 5-22
CLOSE_EVENT_CHN 5-23
INFO_EVENT_CHN 5-24
WAIT_EVENT_CHN 5-25
FLUSH_EVENT_CHN 5-27
SEND_EVENT_CHN 5-28
5.9 Clock System Calls 5-28
DELAY_TIME 5-30
GET_TIME 5-31
SET_LOCAL_TIME_DIFF 5-32
CONVERT_TIME 5-33

EXCEPTIONS AND EVENTS

Processes have several ways to keep informed about the state of the system. Normal process-to-process communication and synchronization employs pipes, shared data segments, or events. Abnormal conditions, including those your program may define, employ exceptions (interrupts). Exceptions are signals, which the process can respond to in a variety of ways under your control.

5.1 Exceptions

Normal execution of a process can be interrupted by an exceptional condition (such as division by zero or reference to an invalid address). Some error conditions are trapped by the hardware and some by the system software. The process itself can define and signal exceptions of your choice.

When an exception occurs, the system first checks the state of the exception. The three exception states are:

- Enabled
- Queued
- Ignored

If the exception is enabled, and defined by the system, the system looks for a user defined handler for that exception. If none is found, the system invokes the default exception handler which usually aborts the process that generated the exception.

If the exception is *enabled* and it was created by the program, the system invokes the associated exception handler. (You create new exceptions by declaring and enabling handlers for the exception.)

If the state of the exception is *queued*, the exception is placed on a queue. When that exception is subsequently enabled, this queue is examined, and the appropriate exception handler is invoked. Processes can flush the exception queue.

If the state of the exception is *ignored*, the system still detects the occurrence of the exception, but the exception is neither honored nor queued. Note that ignoring a system defined exception will have uncertain effects. Although you can cause the system to ignore even the SYS_TERMINATE exception, that capability is provided so that your program can clean up before terminating. You cannot set your program to ignore fatal errors.

Invocation of the exception handler causes the scheduler to run, so it is possible for another process to run between the signalling of the exception and the execution of the exception handler.

5.2 System Defined Exceptions

Certain exceptions are predefined by the Operating System. These include:

- Division by zero (SYS_ZERO_DIV). Default handler aborts process.
- Value out of bounds (that is, range check error) or illegal string index (SYS_VALUE_OOB). Default handler aborts process.
- Arithmetic overflow (SYS_OVERFLOW). Default handler aborts process.
- Process termination (SYS_TERMINATE). This exception is signalled when a process terminates, or when there is a bus error, address error, illegal instruction, privilege violation, or 1111 emulator error. The default handler does nothing. This exception is different from the other system defined exceptions in that the program always terminates as soon as the exception occurs. In the case of other (non-fatal) errors, the program is allowed to continue until the exception is enabled.

Except where otherwise noted, these exceptions are fatal if they occur within Operating System code. The hardware exceptions for parity error, spurious interrupt, and power failure are also fatal.

5.3 Exception Handlers

A user-defined exception handler can be declared for a specific exception. This exception handler is coded as a procedure, but must follow certain conventions. Each handler must have two input parameters: Environment_Ptr and Exception_Ptr. The Operating System ensures that these pointers are valid when the handler is entered. Environment_Ptr points to an area in the stack containing the interrupted environment: register contents, condition flags, and program state. The handler can access this environment and can modify everything except the program counter, register A7, and the supervisor state bit in the status register. The Exception_Ptr points to an area in the stack containing information about the specific exception.

Each exception handler must be defined at the global level of the process, must return, and cannot have any EXIT or global GOTO statements. Because the Operating System disables the exception before calling the exception handler, the handler should re-enable the exception before it returns.

If an exception handler for a given exception already exists when another handler is declared for that exception, the old handler becomes disassociated from the exception.

An exception can occur during the execution of an exception handler. The state of the exception determines whether it is honored, placed on a queue, or ignored. If the second exception has the same name as the exception that is currently being handled and its state is enabled, a nested call to the exception handler occurs. (The system always disables the exception before calling the exception handler, however. Therefore, nested handler calling will only occur if you explicitly enable the exception.)

There is an "exception occurred" flag for every declared exception; it is set whenever the corresponding exception occurs. This flag can be examined and reset. Once the flag is set, it remains set until FLUSH_EXCEP is called.

The following program fragment gives an example of exception handling.

```
PROCEDURE Handler(Env_Ptr:p_env_blk;
                  Data_Ptr:p_ex_data);
VAR ErrNum:INTEGER;
BEGIN
(*Env_Ptr points to a record containing the program counter     *)
(*and all registers. Data_Ptr points to an array of 12 longints *)
(*that contain the event header and text if this handler is     *)
(*associated with an event-call channel (see below)             *)
.
.
.
ENABLE_EXCEP(errnum, excep_name);
.
.
.
END;

BEGIN (*Main program*)
.
.
.
Excep_name:='EndOfDoc';
DECLARE_EXCEP_HDL(errnum, excep_name, @Handler);
.
.
.
SIGNAL_EXCEP(errnum, excep_name, excep_data);
.
.
.
```

At the time the exception handler is invoked, the stack is as shown in Figure 5-1.

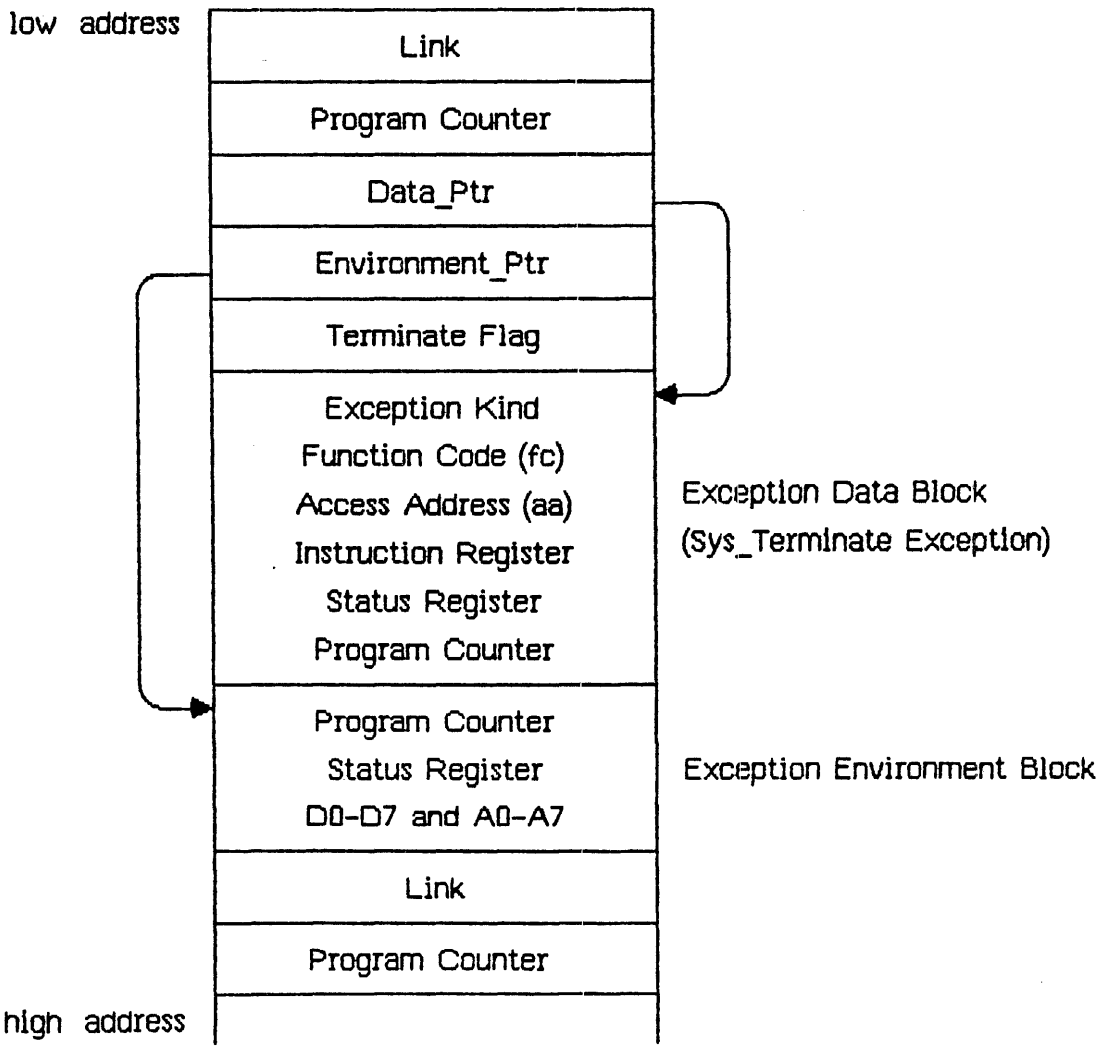

Figure 5-1
Stack at Exception Handler Invocation

The Exception Data Block given here reflects the state of the stack upon a SYS_TERMINATE exception. The term_ex_data record (described in Appendix A) gives the various forms the data block can take. The Excep_kind field (the first, or 0th, longint) gives the cause of the exception. The status register and program counter values in the data block reflect the true (current) state of these

values. The same data in the Environment block reflects the state of these values at the time the exception was signalled, not the values at the time the exception actually occurs.

For SYS_ZERO_DIV, SYS_VALUE_OOB, and SYS_OVERFLOW exceptions, the hard_ex_data record described in Appendix A gives the various forms that the data block can take.

In the case of a bus or address error, the PC (program counter) can be 2 to 10 bytes beyond the current instruction. The PC and A7 cannot be modified by the exception handler.

When a disabled exception is re-enabled, a queued exception may be signalled. In this case, the exception environment reflects the state of the system at the time the exception was re-enabled, not the time at which the exception occurred.

5.4 Events

An event is a piece of information sent by one process to another, generally to help cooperating processes synchronize their activities. An event is sent through a kind of pipe called an event channel. The event is a fixed size data block consisting of a header and some text. The header contains control information; the identifier of the sending process and the type of the event. The header is written by the system, not the sender, and is readable by the receiving process. The event text is written by the sender; its meaning is defined by the sending and receiving processes.

There are several predefined system event types. The predefined type "user" is assigned to all events not sent by the Operating System.

5.5 Event Channels

Event channels can be viewed as higher-level pipes. One important difference is that event channels require fixed size data blocks, whereas pipes can handle an arbitrary byte stream.

An event channel can be defined globally or locally. A global event channel has a globally defined pathname catalogued in the file system, and can be used by any process. A local event channel, however, has no name and is known only by the Operating System and the process that opened it. Local event channels can only be opened by user processes as receivers. A local channel can be opened by the father process to receive system generated events pertaining to its son.

There are two types of global and local event channels: event-wait and event-call. If the receiving process is not ready to receive the event, an event-wait type of event channel queues an event sent to it. An event-call type of event channel, however, forces its event on the process, in effect treating the event as an exception. In that case, an exception name must be given when the event-call event channel is opened, and an exception handler for that exception must be declared. If the process reading the event-call channel is suspended at

the time the event is sent, the event is delivered when the process becomes active.

When an event channel is created, the Operating System preallocates enough space to the channel for typical interprocess communication. If SEND_EVENT_CHN is called when the channel does not have enough space for the event, the calling process is blocked until enough space is freed up.

If WAIT_EVENT_CHN is called when the channel is empty, the calling process is blocked until an event arrives.

The following code fragments use event-wait channels to handle process synchronization. Operating System calls used in these program fragments are documented later in this chapter.

PROCESS A:

```
.
.
.
chn_name := 'event_channel_1';
exception:= '';
receiver := TRUE;
OPEN_EVENT_CHN (errint, chn_name, refnum1, exception, receiver);
chn_name := 'event_channel_2';
receiver := FALSE;
OPEN_EVENT_CHN (errint, chn_name, refnum2, exception, receiver);
waitlist.length := 1;
waitlist.refnum[0] := refnum1;
REPEAT
   event1_ptr^.[0] := agreed_upon_value;
   interval.sec  := 0; (* send event immediately *)
   interval.msec := 0;
   SEND_EVENT_CHN (errint, refnum2, event1_ptr,
      interval, clktime);
   WAIT_EVENT_CHN (errint, waitlist, refnum_signalling,
      event2_ptr);
            .
            .
   (* processing performed here *)
            .
            .
UNTIL AllDone;
.
.
.
```

```
PROCESS B:

      .
      .
      .
      chn_name := 'event_channel_2';
      exception:= '';
      receiver := TRUE;
      OPEN_EVENT_CHN (errint, chn_name, refnum2, exception, receiver);
      chn_name := 'event_channel_1';
      receiver := FALSE;
      OPEN_EVENT_CHN (errint, chn_name, refnum1, exception, receiver);
      waitlist.length := 1;
      waitlist.refnum[0] := refnum1;
      REPEAT
         event2_ptr^.[0] := agreed_upon_value;
         interval.sec  := 0; (* send event immediately *)
         interval.msec := 0;
         WAIT_EVENT_CHN (errint, waitlist, refnum_signalling,
            event1_ptr);
                  .
                  .
         (* processing performed here *)
                  .
                  .
         SEND_EVENT_CHN (errint, refnum2, event2_ptr,
            interval, clktime);
      UNTIL AllDone;
      .
      .
      .
```

The order of execution of the two processes is the same regardless of the process priorities. Process switch always occurs at the WAIT_EVENT_CHN call.

In the following example using event-call channels, process switch may occur at different places in the programs. Process A calls YIELD_CPU, which gives the CPU to Process B only if Process B is ready to run.

```
PROCESS A
     PROCEDURE Handler(Env_ptr:p_env_blk;
                       Data_ptr:p_ex_data);
     .
     .
     .
     BEGIN
        event2_ptr^.[0] := agreed_upon_value;
                  .
                  .
        (* processing performed here *)
                  .
                  .
        interval.sec  := 0; (* send event immediately *)
        interval.msec := 0;
        SEND_EVENT_CHN (errint, refnum2, event2_ptr, interval,
           clktime);
        to_any := true;
        YIELD_CPU (errint, to_any);
     END;

     BEGIN (* Main program*)
     .
     .
     .
     DECLARE_EXCEP_HDL (errint, excep_name_1, entry_point);
     chn_name := 'event_channel_1';
     exception:= excep_name_1;
     receiver := TRUE;
     OPEN_EVENT_CHN (errint, chn_name, refnum1, exception, receiver);
     chn_name := 'event_channel_2';
     receiver := FALSE;
     exception:= '';
     OPEN_EVENT_CHN (errint, chn_name, refnum2, exception, receiver);
     SEND_EVENT_CHN (errint, refnum2, event2_ptr, interval, clktime);
     to_any := true;
     YIELD_CPU (errint,  to_any);
     .
     .
     .
```

```
PROCESS B
     PROCEDURE Handler(Env_ptr:p_env_blk;
                       Data_ptr:p_ex_data);
     .
     .
     .
     BEGIN
        event2_ptr^.[0] := agreed_upon_value;
                  .
                  .
        (* processing performed here *)
                  .
                  .
        interval.sec  := 0; (* send event immediately *)
        interval.msec := 0;
        SEND_EVENT_CHN (errint,refnum1,event2_ptr,interval,
           clktime);
        to_any := true;
        YIELD_CPU (errint,to_any);
     END;
     .
     .
     .
     BEGIN (* Main program *)

     DECLARE_EXCEP_HDL (errint,excep_name_1,entry_point)
     chn_name := 'event_channel_1';
     exception:= excep_name_1;
     receiver := FALSE;
     exception:= '';
     OPEN_EVENT_CHN (errint,chn_name,refnum1,exception,receiver);
     chn_name := 'event_channel_2';
     receiver := TRUE;
     OPEN_EVENT_CHN (errint,chn_name,refnum2,exception,receiver);
     .
     .
     .
     END.
```

5.6 The System Clock

A process can read the system clock time, convert to local time, or delay its own continuation until a given time. The year, month, day, hour, minute, second, and millisecond are available from the clock. The system clock is set up through the Workshop shell (see the *Workshop User's Guide for the Lisa*).

5.7 Exception Management System Calls

This section describes all the Operating System calls that pertain to exception management. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in exception management calls:

```
T_ex_name = STRING[16];
Longadr = ^longint;
T_ex_data = Array [0..11] of longint;
T_ex_sts = Record
             ex_occurred_f:boolean;
             ex_state:t_ex_state;
             num_excep:integer;
             hdl_adr:longadr;
           end;
T_ex_state = (enabled, queued, ignored);
```

```
DECLARE_EXCEP_HDL ( Var ErrNum:Integer;
                    Var Excep_name:t_ex_name;
                        Entry_point:LongAdr)

              ErrNum:      Error indicator
              Excep_name:  Name of exception
              Entry_point: Address of exception handler
```

DECLARE_EXCEP_HDL sets the Operating System so that the occurrence of the exception referred to by excep_name causes the execution of the exception handler at entry_point.

Excep_name is a character string name with up to 16 characters that is locally defined in the process and known only to the process and the Operating System. If entry-point is @NIL, and excep_name specifies a system_exception, the system default exception handler for that exception is used, if it is a system-defined exception. Any previously declared exception handler is disassociated by this call. The exception itself is automatically enabled.

If some excep_name exceptions are queued up at the time of the DECLARE_EXCEP_HDL call, the exception is automatically enabled and the queued exceptions are handled by the newly declared handler.

You can call DECLARE_EXCEP_HDL with an exception handler address of @NIL to disassociate your handler from the exception. If there is no system handler defined, and the program signals the exception, it will receive an error 201.

```
DISABLE_EXCEP (Var ErrNum:Integer;
               Var Excep_name:t_ex_name;
                   Queue:Boolean)
```

```
ErrNum:     Error indicator
Excep_name: Name of exception to be disabled
Queue:      Exception queuing flag
```

A process can explicitly disable the trapping of an exception by calling DISABLE_EXCEP. Excep_name is the name of the exception to be disabled. If queue is true and an exception occurs, the exception is queued and is handled when it is enabled again. If queue is false, the exception is ignored. When an exception handler is entered, the state of the exception in question is automatically set to queued.

If an exception handler is associated through OPEN_EVENT_CHN with an event channel, and DISABLE_EXCEP is called for that exception, then:

- If queue is false, and if an event is sent to the event channel by SEND_EVENT_CHN, the SEND_EVENT_CHN call succeeds, but it is equivalent to not calling SEND_EVENT_CHN at all.
- If queue is true, and if an event is sent to the event channel by SEND_EVENT_CHN, the SEND_EVENT_CHN call succeeds and a call to WAIT_EVENT_CHN will receive the event, thus dequeing the exception.

```
ENABLE_EXCEP (Var ErrNum:Integer;
              Var Excep-name:t_ex_name)
```

```
ErrNum:     Error indicator
Excep_name: Name of exception to be enabled
```

ENABLE_EXCEP causes an exception to be handled again. Since the Operating System automatically disables an exception when its exception handler is entered (see DISABLE_EXCEP), the exception handler should explicitly re-enable the exception before it returns to the process.

```
INFO_EXCEP (Var ErrNum:Integer;
            Var Excep_name:t_ex_name;
            Var Excep_status:t_ex_sts)

          ErrNum:       Error indicator
          Excep_name:   Name of exception
          Excep_Status: Status of exception
```

INFO_EXCEP returns information about the exception specified by excep_name. The parameter excep_status is a record containing information about the exception. This record contains:

```
t_ex_sts = RECORD                  (* exception status       *)
           Ex_occurred_f:Boolean;(*exception occurred flag *)
           Ex_state:t_ex_state;  (* exception status       *)
           Num_excep:integer;(*no. of exceptions queued    *)
           Hdl_adr:Longadr;  (*exception handler's address *)
           END;
```

Once Ex_occurred_f has been set to true, only a call to FLUSH_EXCEP can set it to false.

```
SIGNAL_EXCEP (Var ErrNum:Integer;
              Var Excep_name:t_ex_name;
              Var Excep_data: t_ex_data)

        ErrNum:     Error indicator
        Excep_name: Name of exception to be signalled
        Excep_Data: Information for exception handler
```

A process can signal the occurrence of an exception by calling SIGNAL_EXCEP. The exception handler associated with excep_name is entered. It is passed excep_data, a data area containing information about the nature and cause of the exception. The structure of this information area is:

```
array[0..size_exdata] of Longint
```

SIGNAL_EXCEP can be used for user-defined exceptions, and for testing exception handlers defined to handle system-defined exceptions.

```
FLUSH_EXCEP (Var ErrNum:Integer;
             Var Excep_name:t_ex_name)

        ErrNum:     Error indicator
        Excep_name: Name of exception whose queue is flushed
```

FLUSH_EXCEP clears out the queue associated with the exception excep_name and resets its "exception occurred" flag.

5.8 Event Management System Calls

This section describes all the Operating System calls that pertain to event management. A summary of all the Operating System calls can be found in Appendix A. The following special types are used in event management calls:

```
Pathname = STRING[255];
T_ex_name = STRING[16];
T_chn_sts = Record
              chn_type:chn_kind;
              num_events:integer;
              open_recv:integer;
              open_send:integer;
              ec_name:pathname;
            end;
chn_kind = (wait_ec, call_ec);
T_waitlist = Record
               length:integer;
               refnum:array [0..10] of integer;
             end;
P_r_eventblk = ^r_eventblk;
R_eventblk = Record
               event_header:t_eheader;
               event_text:t_event_text;
             end;
T_eheader = Record
              send_pid:longint;
              event_type:longint;
            end;
T_event_text = array [0..9] of longint;
P_s_eventblk = ^s_eventblk;
S_eventblk = T_event_text;
Timestmp_interval = Record
                      sec:longint;
                      msec:0..999;
                    end;
```

```
Time_rec = Record
             year:integer;
             day:1..366;
             hour:-23..23;
             minute:-59..59;
             second:0..59;
             msec:0..999;
           end;
```

```
MAKE_EVENT_CHN (Var ErrNum:Integer;
                Var Event_chn_name:Pathname)

        ErrNum:          Error indicator
        Event_chn_name: Pathname of event channel
```

MAKE_EVENT_CHN creates an event channel with the name given in event_chn_name. The name must be a file system pathname; it cannot be null.

```
KILL_EVENT_CHN (Var ErrNum:Integer;
                Var Event_chn_name:Pathname)

        ErrNum:          Error indicator
        Event_chn_name:  Pathname of event channel
```

To delete an event channel, call KILL_EVENT_CHN. The actual deletion is delayed until all processes using the event channel have closed it. In the period between the KILL_EVENT_CHN call and the channel's actual deletion, no processes can open it. A channel can be deleted by any process that knows the channel's name.

```
OPEN_EVENT_CHN (Var ErrNum:Integer;
                Var Event_chn_name:Pathname;
                Var Refnum:Integer;
                    Excep_name:t_ex_name;
                    Receiver:Boolean)
```

ErrNum:	Error indicator
Event_chn_name:	Pathname of event channel
RefNum:	Identifier of event channel
Excep_name:	Exception name, if any
Receiver:	Access mode of calling process

OPEN_EVENT_CHN opens an event channel and defines its attributes from the process point of view. Refnum is returned by the Operating System to be used in any further references to the channel.

Event_chn_name determines whether the event channel is locally or globally defined. If it is a null string, the event channel is locally defined. If event_chn_name is not null, it is the file system pathname of the channel.

Excep_Name determines whether the channel is an event-wait or event-call channel. If it is a null string, the channel is of event-wait type. Otherwise, the channel is an event-call channel and excep_name is the name of the exception that is signalled when an event arrives in the channel. The excep_name must be declared before its use in the OPEN_EVENT_CHN call.

Receiver is a boolean value indicating whether the process is opening the channel as a sender (receiver is false) or a receiver (receiver is true). A local channel (one with a null pathname) can be opened only to receive events. Also, a call-type channel can only be opened as a receiver.

```
CLOSE_EVENT_CHN (Var ErrNum:Integer;
                     Refnum:Integer)
```

```
ErrNum: Error indicator
Refnum: Identifier of event channel to be closed
```

CLOSE_EVENT_CHN closes the event channel associated with refnum. Any events queued in the channel remain there. The channel cannot be accessed until it is opened again.

If the channel has previously been killed with KILL_EVENT_CHN, you will not be able to open it after it has been closed.

If the channel has not been killed, it can be opened by OPEN_EVENT_CHN.

```
INFO_EVENT_CHN (Var ErrNum:Integer;
                    Refnum:Integer;
                Var Chn_Info:t_chn_sts)
```

ErrNum: Error indicator
Refnum: Identifier of event channel
Chn_Info: Status of event channel

INFO_EVENT_CHN gives a process information about an event channel. The Operating System returns a record, chn_info, with information pertaining to the channel associated with refnum.

The definition of the type of the chn_info record is:

```
t_chn_sts =
     RECORD                  (* event channel status     *)
     Chn_type:Chn_kind;      (* wait_ec or call_ec       *)
     Num_events:Integer;     (* number of queued events  *)
     Open_recv:Integer;      (* number of processes reading
                                channel                  *)
     Open_send:integer;      (* no. of processes sending to
                                this channel             *)
     Ec_name:pathname;       (* event channel name       *)
     END;
```

```
WAIT_EVENT_CHN (Var ErrNum:Integer;
                Var Wait_List:t_waitlist;
                Var RefNum:Integer;
                    Event_ptr:p_r_eventblk)
```

ErrNum:	Error indicator
Wait_list:	Record with array of event channel refnums
Refnum:	Identifier of channel that had an event
Event_ptr:	Pointer to event data

WAIT_EVENT_CHN puts the calling process in a waiting state pending the arrival of an event in one of the specified channels. Wait_list is a pointer to a list of event channel identifiers. When an event arrives in any of these channels, the process is made ready to execute. Refnum identifies which channel got the event, and event_ptr points to the event itself.

A process can wait for any boolean combination of events. If it must wait for any event from a set of channels (an OR condition), it should call WAIT_EVENT_CHN with wait_list containing the list of event channel identifiers. If, on the other hand, it must wait for all the events from a set of channels (an AND condition), then for each channel in the set, WAIT_EVENT_CHN should be called with a wait_list containing just that channel identifier.

The structure of t_waitlist is:

```
RECORD
        Length:Integer;
        Refnum:Array[0..size_waitlist] of Integer;
END;
```

Event_ptr is a pointer to a record containing the event header and the event text. Its definition is:

```
P_r_eventblk = ^r_eventblk;
R_eventblk = Record
                event_header:t_eheader;
                event_text:t_event_text;
             end;
T_eheader = Record
                send_pid:longint;
                event_type:longint;
            end;
T_event_text = array [0..9] of longint;
```

Send_pid is the process id of the sender.

Currently, the possible event type values are:

```
1   =   Event sent by user process
2   =   Event sent by system
```

When you receive the SYS_SON_TERM event, the first longint of the event text contains the termination cause of the son process. The cause is same as that given in the SYS_TERMINATE exception given to the son process. The rest of the event text can be filled by the son process.

If you call WAIT_EVENT_CHN on an event-call channel that has queued events, the event is treated just like an event in an event-wait channel. If WAIT_EVENT_CHN is called on an event-call channel that does not have any queued events, an error is returned.

```
FLUSH_EVENT_CHN (Var ErrNum:Integer;
                     Refnum:Integer)
```

ErrNum: Error indicator
Refnum: Identifier of event channel to be flushed

FLUSH_EVENT_CHN clears out the specified event channel. All events queued in the channel are removed. If this is called by a sender, it has no effect.

```
SEND_EVENT_CHN (Var ErrNum:Integer;
                    Refnum:Integer;
                    Event_ptr:p_s_eventblk;
                    Interval:Timestmp_interval;
                    Clktime:Time_rec)

        ErrNum:       Error indicator
        Refnum:       Channel for event
        Event_ptr:    Pointer to event data
        Interval:     Timer for event
        Clktime:      time data for event
```

SEND_EVENT_CHN sends an event to the channel specified by refnum. Event_ptr points to the event that is to be sent. The event data area contains only the event text; the header is added by the system.

If the event is of the event-wait type, the event is queued. Otherwise the Operating System signals the corresponding exception for the process receiving the event.

If the channel is open by several senders, the receiver can sort the events by the process identifier which the Operating System places in the event header. Alternatively, the senders can place predefined identifiers in the event text which identify the sender.

The parameter Interval, indicates whether the event is a timed event. Timestmp_interval is a record containing a second and a millisecond field. If both fields are 0, the event is sent immediately. If the second given is less than 0, the millisecond field is ignored and the time_rec record is used. If the time in the time_rec has already passed, the event is sent immediately. If the millisecond field is greater than 0, and the second field is greater than or equal to 0, the event is sent that number of seconds and milliseconds from the present.

A process can time out a request to another process by sending itself a timed event and then waiting for the arrival of either the timed event or an event indicating the request has been served. If the timed event is received first, the request has timed out. A process can also time its own progress by periodically sending itself a timed event through an event-call event channel.

5.9 Clock System Calls

This section describes all the Operating System calls that pertain to the clock. A summary of all the Operating System calls can be found in Appendix A.

The following special types are used in clock calls:

```
Timestmp_interval = Record
                       sec:longint;
                       msec:0..999;
                    end;
Time_rec = Record
             year:integer;
             day:1..366;
             hour:-23..23;
             minute:-59..59;
             second:0..59;
             msec:0..999;
           end;
Hour_range = -23..23
Minute_range = -59..59;
```

```
DELAY_TIME (Var ErrNum:Integer;
                Interval:T_interval;
                Clktime:Time_rec)

          ErrNum:   Error indicator
          Interval: Delay timer
          Clktime:  Time information
```

DELAY_TIME stops execution of the calling process for the number of SECONDS and milliseconds specified in the interval record. If this time period is zero, DELAY_TIME has no effect. If the period is less than zero, execution of the process is delayed until the time specified by Clktime.

```
GET_TIME (Var ErrNum:Integer;
          Var Sys_Time:Time_rec)

          ErrNum:   Error indicator
          Sys_Time: Time information
```

GET_TIME returns the current system clock time in the record Sys_Time. The msec field of Sys_Time always contains a 0 on return.

```
SET_LOCAL_TIME_DIFF (Var ErrNum:Integer;
                         Hour:Hour_range;
                         Minute:Minute_range)
```

ErrNum: Error indicator
Hour: Number of hours difference from the system clock
Minute: Number of minutes difference from the system clock

SET_LOCAL_TIME_DIFF informs the Operating System of the difference in hours and minutes between the local time and the system clock. Hour and Minute can be negative.

```
CONVERT_TIME (Var ErrNum:Integer;
              Var Sys_Time:Time_rec;
              Var Local_Time:Time_rec;
                  To_sys:Boolean)
```

```
ErrNum:      Error indicator
Sys_Time:    System clock time
Local_Time:  Local time
To_sys:      Direction of time conversion
```

CONVERT_TIME converts between local time and system clock time.

To_sys is a boolean value indicating which direction the conversion is to go. If it is true, the system takes the time data in local_time and puts the corresponding system time in Sys_Time. Otherwise, it takes the time data in Sys_Time and puts the corresponding local time in local_time. Both time data areas contain the year, month, day, hour, minute, second, and millisecond.

Chapter 6
CONFIGURATION

6.1 Configuration System Calls 6-3
CARDS_EQUIPPED 6-4
GET_CONFIG_NAME 6-5
OSBOOTVOL 6-6

CONFIGURATION

Every Lisa system is configured using the Preferences tool. Preferences places the configuration state of the system in a special part of the system's memory called *parameter memory*. Although parameter memory is not contained on a disk, it is supplied with battery power, so that the contents are kept even when the system is turned off. Note that the batteries are charged as long as Lisa is plugged in, even if the unit is powered off. Also, the batteries will keep parameter memory secured for several hours, even if line power is lost. In addition, every time parameter memory is changed, a copy of the new data is made on the boot disk. If the contents of parameter memory are lost, this disk copy is automatically restored to parameter memory.

Since the devices actually connected may differ from the configuration stored in parameter memory, three calls are provided that allow programs to request some information about the configuration of the system.

In addition, two calls are provided to directly read and write the contents of parameter memory.

6.1 Configuration System Calls

This section describes all the Operating System calls that pertain to configuration. A summary of all the Operating System calls can be found in Appendix A. Special data types used by configuration calls are defined along with the calls.

```
CARDS_EQUIPPED (Var Errnum:Integer;
                Var In_slot:Slot_array)
```

Errnum: Error code
In_slot: Identifies the types of cards configured

This call returns an array showing the types of cards which are in the various card slots.

The definition of Slot_array is:

```
slot_array = array [1..3] of card_types;
```

where:

```
card_types = (no_card,
              apple_card,
              n_port_card,
              net_card,
              laser_card);
```

```
GET_CONFIG_NAME (Var Errnum:Integer;
                     Devpostn:Tports;
                 Var Devname:E_name)
```

Errnum: Error code
Devpostn: A port identifier
Devname: The name of the device attached to the port

This call returns the name of the device configured at the port given in devpostn. See **OSBOOTVOL** for the definition of tports. Type e_name is defined as:

```
E_name = STRING [32];
```

OSBOOTVOL (Var Errnum:Integer) : Tports

Errnum: Error code
Tports: Identifies the port to which the boot volume is attached

OSBOOTVOL is a function that returns the identifier for the port attached to the boot volume. Note that this port might not be the port configured for the boot volume, since it is possible for the user to override the default boot. Note also that the port identifier is not the same as the device name. You can use GET_CONFIG_NAME to find out the name of the device attached to the port.

Tports is a set that has this definition:

```
tports = (uppertwig, lowertwig, parallel,
          slot11, slot12, slot13, slot14,
          slot21, slot22, slot23, slot24,
          slot31, slot32, slot33, slot34,
          serialA, serialB, main_console, alt_console,
          t_mouse,    t_speaker,    t_extra1,    t_extra2,
          t_extra3);
```

Appendix A
OPERATING SYSTEM INTERFACE

```
UNIT syscall;                    (* system call definitions unit *)
INTRINSIC;

INTERFACE

CONST
max_ename = 32;          (* maximum length of a file system object name    *)
max_pathname = 255;      (* maximum length of a file system pathname       *)
max_label_size = 128;    (* maximum size of a file label, in bytes         *)
len_exname = 16;         (* length of exception name                       *)
                    size_exdata = 11;       (* 48 bytes, exception data
                    block should have the same size as r_eventblk, received
                    event block                                            *)

size_etext = 9;          (* event text size - 40 bytes                     *)
size_waitlist = 10; (* size of wait list - should be same as reqptr_list  *)

(* exception kind definitions for 'SYS_TERMINATE' exception                *)
call_term = 0;             (* process called terminate_process            *)
ended     = 1;             (* process executed 'end' statement            *)
self_killed = 2;           (* process called kill_process on self         *)
killed      = 3;           (* process was killed by another process       *)
fthr_term   = 4;           (* process's father is terminating             *)
bad_syscall = 5;           (* process made invalid sys call - subcode bad *)
bad_errnum  = 6;           (* process passed bad address for errnum parm  *)
swap_error  = 7;           (* process aborted due to code swap-in error   *)
stk_overflow = 8;          (* process exceeded max size (+T nnn) of stack *)
data_overflow = 9;         (* process tried to exceed max data space size *)
parity_err  = 10;          (* process got a parity error while executing  *)

def_div_zero   = 11;(* default handler for div zero exception was called *)
def_value_oob  = 12;       (* " for value oob exception                   *)
def_ovfw       = 13;       (* " for overflow exception                    *)
def_nmi_key    = 14;       (* " for NMI key exception                     *)
def_range      = 15;(*" for 'SYS_VALUE_OOB' excep due to value range err  *)
def_str_index  = 16;(*" for 'SYS_VALUE_OOB' excep due to string index err *)

bus_error = 21;            (* bus error occurred                          *)
addr_error = 22;           (* address error occurred                      *)
illg_inst = 23;            (* illegal instruction trap occurred           *)
```

```
priv_violation = 24;          (* privilege violation trap occurred          *)
line_1010 = 26;               (* line 1010 emulator occurred                *)
line_1111 = 27;               (* line 1111 emulator occurred                *)

unexpected_ex = 29;           (* an unexpected exception occurred           *)

div_zero    = 31;    (* exception kind definitions for hardware exception   *)
value_oob   = 32;
ovfw        = 33;
nmi_key     = 34;
value_range = 35;    (* excep kind for value range and string index error   *)
str_index   = 36;    (* Note that these two cause 'SYS_VALUE_OOB' excep     *)

(*DEVICE_CONTROL functions*)

dvParity = 1;                 (*RS-232*)
dvOutDTR = 2;                 (*RS-232*)
dvOutXON = 3;                 (*RS-232*)
dvOutDelay = 4;               (*RS-232*)
dvBaud = 5;                   (*RS-232*)
dvInWait = 6;                 (*RS-232, CONSOLE*)
dvInDTR = 7;                  (*RS-232*)
dvInXON = 8;                  (*RS-232*)
dvTypeahd = 9;                (*RS-232*)
dvDiscon = 10;                (*RS-232*)
dvOutNoHS = 11;               (*RS-232*)
dvErrStat = 15;               (*PROFILE*)
dvGetEvent = 16;              (*CONSOLE*)
dvAutoLF = 17;                (*RS-232, CONSOLE, PARALLEL PRINTER*) (*not yet*)
dvDiskStat = 20;              (*DISKETTE, PROFILE*)
dvDiskSpare = 21;             (*DISKETTE, PROFILE*)


TYPE
pathname = string [max_pathname];
e_name = string [max_ename];
namestring = string [20];
procinfoRec = record
progpathname : pathname;
global_id    : longint;
father_id    : longint;
priority     : 1..255;
state        : (pactive, psuspended, pwaiting);
data_in      : boolean
end;
```

```
Tdstype = (ds_shared, ds_private); (* types of data segments *)

dsinfoRec = record
mem_size : longint;
disc_size: longint;
numb_open : integer;
ldsn : integer;
boundF : boolean;
presentF : boolean;
creatorF : boolean;
rwaccess : boolean;
segptr : longint;
volname: e_name;
end;


t_ex_name = string [len_exname];                (* exception name      *)
longadr = ^longint;
t_ex_state = (enabled, queued, ignored);        (* exception state     *)
p_ex_data = ^t_ex_data;
t_ex_data = array [0..size_exdata] of longint; (* exception data blk *)
t_ex_sts = record                               (* exception status    *)
ex_occurred_f : boolean;    (* exception occurred flag   *)
ex_state : t_ex_state;      (* exception state           *)
num_excep : integer;        (* number of exceptions q'ed *)
hdl_adr : longadr;          (* handler address           *)
end;
p_env_blk = ^env_blk;
env_blk = record            (* environment block to pass to handler *)
pc : longint;               (* program counter                      *)
sr : integer;               (* status register                      *)
d0 : longint;               (* data registers 0 - 7                 *)
d1 : longint;
d2 : longint;
d3 : longint;
d4 : longint;
d5 : longint;
d6 : longint;
d7 : longint;
a0 : longint;               (* address registers 0 - 7 *)
a1 : longint;
a2 : longint;
a3 : longint;
a4 : longint;
```

```
a5 : longint;
a6 : longint;
a7 : longint;
end;

p_term_ex_data = ^term_ex_data;
term_ex_data = record              (* terminate exception data block *)
case excep_kind : longint of
call_term,
ended,
self_killed,
killed,
fthr_term,
bad_syscall,
bad_errnum,
swap_error,
stk_overflow,
data_overflow,
parity_err : ();   (* due to process termination                      *)

illg_inst,
priv_violation,    (* due to illegal instruction, privilege violation  *)
line_1010,
line_1111,         (* due to line 1010, 1111 emulator                  *)
def_div_zero,
def_value_oob,
def_ovfw,
def_nmi_key        (* terminate due to default handler for hardware
                      exception                                        *)
  : (sr : integer;
  pc : longint);  (* at the time of occurrence                        *)
def_range,
def_str_index    (* terminate due to default handler for 'SYS_VALUE_OOB' excep for
                      value range or string index error               *)
  : (value_check : integer;
  upper_bound : integer;
  lower_bound : integer;
  return_pc   : longint;
  caller_a6   : longint);
bus_error,
addr_error                         (* due to bus error or address error  *)
  : (fun_field : packed record     (* one integer                         *)
filler : 0..$7ff;                  (* 11 bits                             *)
r_w_flag : boolean;
i_n_flag : boolean;
```

```
fun_code : 0..7; (* 3 bits *)
end;
access_adr : longint;
inst_register : integer;
sr_error : integer;
pc_error : longint);
end;

p_hard_ex_data = ^hard_ex_data;
hard_ex_data = record                    (* hardware exception data block *)
case excep_kind : longint of
div_zero, value_oob, ovfw
: (sr : integer;
pc : longint);
value_range, str_index
: (value_check : integer;
upper_bound : integer;
lower_bound : integer;
return_pc   : longint;
caller_a6   : longint);
end;


accesses = (dread, dwrite, append, private, global_refnum);
mset = set of accesses;
iomode = (absolute, relative, sequential);

UID = record  (*unique id*)
a,b: longint
end;

timestmp_interval = record        (* time interval                          *)
sec : longint;                    (* number of seconds                      *)
msec : 0..999;                   (*number of milliseconds within a second  *)
end;

info_type = (device_t, volume_t, object_t);
devtype = (diskdev, pascalbd, seqdev, bitbkt, non_io);
filetype = (undefined, MDDFfile, rootcat, freelist, badblocks, sysdata,
                    spool, exec, usercat, pipe, bootfile, swapdata,
                    swapcode, ramap, userfile, killedobject);

entrytype= (emptyentry, catentry, linkentry, fileentry, pipeentry, ecentry,
                    killedentry);
```

```
fs_info = record
name : e_name;
dir_path : pathname;
machine_id : longint;
fs_overhead : integer;
result_scavenge : integer;
case otype : info_type of
device_t, volume_t: (
iochannel : integer;
devt : devtype;
slot_no : integer;
fs_size : longint;
vol_size : longint;
blockstructured, mounted : boolean;
opencount : longint;
privatedev, remote, lockeddev : boolean;
mount_pending, unmount_pending : boolean;
volname, password : e_name;
fsversion, volnum : integer;
volid : UID;
backup_volid : UID;
blocksize, datasize, clustersize, filecount : integer;
label_size : integer;
freecount : longint;
DTVC, DTCC, DTVB, DTVS : longint;
master_copy_id, copy_thread : longint;
overmount_stamp : UID;
boot_code : integer;
boot_environ : integer;
privileged, write_protected : boolean;
master, copy, copy_flag, scavenge_flag : boolean;
vol_left_mounted : boolean );

object_t : (
size : longint;
psize : longint;          (* physical file size in bytes          *)
lpsize : integer;         (* logical page size in bytes for this file *)
ftype : filetype;
etype : entrytype;
DTC, DTA, DTM, DTB, DTS : longint;
refnum : integer;
fmark : longint;
acmode : mset;
nreaders, nwriters, nusers : integer;
fuid : UID;
```

```
user_type : integer;
user_subtype : integer;
system_type : integer;
eof, safety_on, kswitch : boolean;
private, locked, protected, master_file : boolean;
file_scavenged, file_closed_by_OS, file_left_open:boolean)
end;

dctype = record
dcversion : integer;
dccode : integer;
dcdata : array [0..9] of longint;          (* user/driver defined data  *)
end;

t_waitlist = record                        (* wait list                 *)
length : integer;
refnum : array [0..size_waitlist] of integer;
end;

t_eheader = record                         (* event header              *)
send_pid : longint;                        (* sender's process id       *)
event_type : longint;                      (* type of event             *)
end;

t_event_text = array [0..size_etext] of longint;
p_r_eventblk = ^r_eventblk;
r_eventblk = record
event_header : t_eheader;
event_text : t_event_text;
end;

p_s_eventblk = ^s_eventblk;
s_eventblk = t_event_text;

time_rec = record
year : integer;
day : 1..366;                  (* julian date   *)
hour : -23..23;
minute : -59..59;
second : 0..59;
msec : 0..999;
end;

chn_kind = (wait_ec, call_ec);
t_chn_sts = record             (* channel status *)
```

```
chn_type : chn_kind;            (* channel type                  *)
num_events : integer;           (* number of events queued       *)
open_recv : integer;            (* number of opens for receiving *)
open_send : integer;            (* number of opens for sending   *)
ec_name : pathname;             (* event channel name            *)
end;

hour_range = -23..23;
minute_range = -59..59;


 {configuration stuff: }

tports = (uppertwig, lowertwig, parallel,
slot11, slot12, slot13, slot14,
slot21, slot22, slot23, slot24,
slot31, slot32, slot33, slot34,
serialA, serialB, main_console, alt_console,
t_mouse, t_speaker, t_extra1, t_extra2, t_extra3);

card_types = (no_card, apple_card, n_port_card, net_card, laser_card);

slot_array = array [1..3] of card_types;


 { Lisa Office System parameter memory type }

pmByteUnique = -128..127;
pMemRec = array[1..62] of pmByteUniqueForAllTheDamnCryBabies;



(* File System calls *)

procedure MAKE_FILE (var ecode:integer; var path:pathname;
                     label_size:integer);

procedure MAKE_PIPE (var ecode:integer; var path:pathname;
                     label_size:integer);

procedure MAKE_CATALOG (var ecode:integer; var path:pathname;
                     label_size:integer);
```

```
procedure MAKE_LINK (var ecode:integer; var path, ref:pathname;
                     label_size:integer);

procedure KILL_OBJECT (var ecode:integer; var path:pathname);

procedure  UNKILL_FILE (var ecode:integer; refnum:integer; var
                     new_name:e_name);

procedure OPEN (var ecode:integer; var path:pathname; var refnum:integer;
                     manip:mset);

procedure CLOSE_OBJECT (var ecode:integer; refnum:integer);

procedure READ_DATA (var ecode : integer;
refnum : integer;
data_addr : longint;
count : longint;
var actual : longint;
mode : iomode;
offset : longint);

procedure WRITE_DATA (var ecode : integer;
refnum : integer;
data_addr : longint;
count : longint;
var actual : longint;
mode : iomode;
offset : longint);

procedure FLUSH (var ecode:integer; refnum:integer);

procedure LOOKUP (var ecode : integer;
var path : pathname;
var attributes : fs_info);

procedure INFO (var ecode:integer; refnum:integer; var refinfo:fs_info);

procedure ALLOCATE (var ecode : integer;
refnum : integer;
contiguous : boolean;
count : longint;
var actual : longint);
```

```
procedure TRUNCATE (var ecode : integer;  refnum : integer);

procedure COMPACT (var ecode : integer;  refnum : integer);

procedure RENAME_ENTRY ( var ecode:integer; var path:pathname; var newname :
                    e_name );

procedure READ_LABEL (  var ecode : integer;
var path : pathname;
data_addr : longint;
count : longint;
var actual : longint  );

procedure WRITE_LABEL (  var ecode : integer;
var path : pathname;
data_addr : longint;
count : longint;
var actual : longint  );

procedure MOUNT ( var ecode:integer; var vname : e_name; var password :
                    e_name ;var devname : e_name);

procedure UNMOUNT ( var ecode:integer; var vname : e_name );

procedure SET_WORKING_DIR ( var ecode:integer; var path:pathname );

procedure GET_WORKING_DIR ( var ecode:integer; var path:pathname );

procedure SET_SAFETY (var ecode:integer;var path:pathname;on_off:boolean );

procedure DEVICE_CONTROL ( var ecode:integer; var path:pathname;
      var cparm : dctype );

procedure RESET_CATALOG (var ecode : integer; var path : pathname);

procedure GET_NEXT_ENTRY (var ecode : integer; var prefix, entry : e_name);

procedure SET_FILE_INFO ( var ecode  : integer;
     refnum : integer;
     fsi    : fs_info );
```

```
(* Process Management system calls *)

function My_ID : longint;

procedure Info_Process (var errnum : integer;  proc_id : longint;
   var proc_info : procinfoRec);

procedure Yield_CPU (var errnum : integer;  to_any : boolean);

procedure SetPriority_Process (var errnum : integer;  proc_id : longint;
new_priority : integer);

procedure Suspend_Process (var errnum : integer;  proc_id : longint;
susp_family : boolean);

procedure Activate_Process (var errnum : integer;  proc_id : longint;
act_family : boolean);

procedure Kill_Process (var errnum : integer;  proc_id : longint);

procedure Terminate_Process (var errnum : integer;  event_ptr :
                    p_s_eventblk);

procedure Make_Process (var errnum : integer;  var proc_id : longint;
var progfile : pathname;  var entryname : namestring;
evnt_chn_refnum : integer);



(* Memory Management system calls *)

procedure make_dataseg(var errnum: integer; var segname: pathname;
mem_size, disc_size: longint; var refnum: integer;
var segptr: longint; ldsn: integer; dstype: Tdstype);

procedure kill_dataseg (var errnum : integer; var segname : pathname);

procedure open_dataseg (var errnum : integer; var segname : pathname;
var refnum : integer; var segptr : longint;
ldsn : integer);

procedure close_dataseg (var errnum : integer; refnum : integer);
```

```
procedure size_dataseg (var errnum : integer; refnum : integer;
deltamemsize : longint; var newmemsize : longint;
deltadiscsize: longint; var newdiscsize: longint);

procedure info_dataseg (var errnum : integer; refnum : integer;
var dsinfo : dsinfoRec);

procedure setaccess_dataseg (var errnum : integer; refnum : integer;
readonly : boolean);

procedure unbind_dataseg (var errnum : integer; refnum : integer);

procedure bind_dataseg(var errnum : integer; refnum : integer);

procedure info_ldsn (var errnum : integer; ldsn: integer; var refnum:
                     integer);

procedure flush_dataseg(var errnum: integer; refnum: integer);

procedure mem_info(var errnum: integer;
var swapspace, dataspace,
cur_codesize, max_codesize: longint);

procedure info_address(var errnum: integer; address: longint;
var refnum: integer);


(* Exception Management system calls *)


procedure declare_excep_hdl (var errnum : integer;
var excep_name : t_ex_name;
entry_point : longadr);

procedure disable_excep (var errnum : integer;
var excep_name : t_ex_name;
queue : boolean);

procedure enable_excep (var errnum : integer;
var excep_name : t_ex_name);

procedure signal_excep (var errnum : integer;
var excep_name : t_ex_name;
excep_data : t_ex_data);
```

```
procedure info_excep (var errnum : integer;
var excep_name : t_ex_name;
var excep_status : t_ex_sts);

procedure flush_excep (var errnum : integer;
var excep_name : t_ex_name);


(* Event Channel management system calls *)

procedure make_event_chn (var errnum : integer;
var event_chn_name : pathname);

procedure kill_event_chn (var errnum : integer;
var event_chn_name : pathname);

procedure open_event_chn (var errnum : integer;
var event_chn_name : pathname;
var refnum : integer;
var excep_name : t_ex_name;
receiver : boolean);

procedure close_event_chn (var errnum : integer;
refnum : integer);

procedure info_event_chn (var errnum : integer;
refnum : integer;
var chn_info : t_chn_sts);

procedure wait_event_chn (var errnum : integer;
var wait_list : t_waitlist;
var refnum : integer;
event_ptr : p_r_eventblk);

procedure flush_event_chn (var errnum : integer;
refnum : integer);

procedure send_event_chn (var errnum : integer;
refnum : integer;
event_ptr : p_s_eventblk;
interval : timestmp_interval;
clktime : time_rec);
```

```
(* Timer functions system calls *)

procedure delay_time (var errnum : integer;
interval : timestmp_interval;
clktime : time_rec);

procedure get_time (var errnum : integer;
var gmt_time : time_rec);

procedure set_local_time_diff (var errnum : integer;
hour : hour_range;
minute : minute_range);

procedure convert_time (var errnum : integer;
var gmt_time : time_rec;
var local_time : time_rec;
to_gmt : boolean);


{configuration stuff}


function OSBOOTVOL(var error : integer) : tports;

procedure GET_CONFIG_NAME( var error    : integer;
devpostn : tports;
var devname  : e_name);

procedure CARDS_EQUIPPED(var error    : integer;
var in_slot : slot_array);


IMPLEMENTATION


procedure MAKE_FILE;  external;

procedure MAKE_PIPE;  external;

procedure MAKE_CATALOG;  external;

procedure MAKE_LINK;  external;

procedure KILL_OBJECT;  external;
```

```
procedure OPEN; external;
procedure CLOSE_OBJECT; external;
procedure READ_DATA; external;
procedure WRITE_DATA; external;
procedure FLUSH; external;
procedure LOOKUP; external;
procedure INFO; external;
procedure ALLOCATE; external;
procedure TRUNCATE; external;
procedure COMPACT; external;
procedure RENAME_ENTRY; external;
procedure READ_LABEL; external;
procedure WRITE_LABEL; external;
procedure MOUNT; external;
procedure UNMOUNT; external;
procedure SET_WORKING_DIR; external;
procedure GET_WORKING_DIR; external;
procedure SET_SAFETY; external;
procedure DEVICE_CONTROL; external;
procedure RESET_CATALOG; external;
procedure GET_NEXT_ENTRY; external;
procedure GET_DEV_NAME; external;
```

```
function My_ID; external;
procedure Info_Process; external;
procedure Yield_CPU; external;
procedure SetPriority_Process; external;
procedure Suspend_Process; external;
procedure Activate_Process; external;
procedure Kill_Process; external;
procedure Terminate_Process; external;
procedure Make_Process; external;
procedure Sched_Class; external;

procedure make_dataseg; external;
procedure kill_dataseg; external;
procedure open_dataseg; external;
procedure close_dataseg; external;
procedure size_dataseg; external;
procedure info_dataseg; external;
procedure setaccess_dataseg; external;
procedure unbind_dataseg; external;
procedure bind_dataseg; external;
procedure info_ldsn; external;
procedure flush_dataseg; external;
procedure mem_info; external;
```

```
procedure declare_excep_hdl; external;
procedure disable_excep; external;
procedure enable_excep; external;
procedure signal_excep; external;
procedure info_excep; external;
procedure flush_excep; external;

procedure make_event_chn; external;
procedure kill_event_chn; external;
procedure open_event_chn; external;
procedure close_event_chn; external;
procedure info_event_chn; external;
procedure wait_event_chn; external;
procedure flush_event_chn; external;
procedure send_event_chn; external;

procedure delay_time; external;
procedure get_time; external;
procedure set_local_time_diff; external;
procedure convert_time; external;
procedure set_file_info; external;
function ENABLEDBG; external;
function OSBOOTVOL; external;
```

```
procedure GET_CONFIG_NAME; external;
function DISK_LIKELY; external;
procedure CARDS_EQUIPPED; external;
procedure Read_PMem;  external;
procedure Write_PMem;  external;
end.
```

Appendix B
SYSTEM RESERVED EXCEPTION NAMES

SYS_OVERFLOW	overflow exception. Signalled if the TRAPV instruction is executed, and the overflow condition is on.
SYS_VALUE_OOB	value out of bound exception. Signalled if the CHK instruction is executed, and the value is less than 0 or greater than upper bound.
SYS_ZERO_DIV	division by zero exception. Signalled if the DIVS or DIVU instruction is executed, and the divisor is zero.
SYS_TERMINATE	termination exception. Signalled when a process is to be terminated.

Appendix C
SYSTEM RESERVED EVENT TYPES

SYS_SON_TERM	"son terminate" event type. If a father process has created a son process with a local event channel, this event is sent to the father process when the son process terminates.

Appendix D
OPERATING SYSTEM ERROR MESSAGES

-1885 Profile not present during driver initialization
-1882 Profile not present during driver initialization
-1176 Data in the object has been altered by Scavenger
-1175 File or volume was scavenged
-1174 File was left open or volume was left mounted, and system crashed
-1173 File was last closed by the OS
-1146 Only a portion of the space requested was allocated
-1063 Attempt to mount boot volume from another Lisa or not most recent boot volume
-1060 Attempt to mount a foreign boot disk following a temporary unmount
-1059 The bad block directory of the diskette is almost full or difficult to read
-696 Printer out of paper during initialization
-660 Cable disconnected during Profile initialization
-626 Scavenger indicated data is questionable, but may be OK
-622 Parameter memory and the disk copy were both invalid
-621 Parameter memory was invalid but the disk copy was valid
-620 Parameter memory was valid but the disk copy was invalid
-413 Event channel was scavenged
-412 Event channel was left open and system crashed
-321 Data segment open when the system crashed. Data possibly invalid.
-320 Could not determine size of data segment
-150 Process was created, but a library used by program has been scavenged & altered
-149 Process was created, but the specified program file has been scavenged & altered
-125 Sepcified process is already terminating
-120 Specified process is already active
-115 Specified process is already suspended
100 Specified process does not exist
101 Specified process is a system process
110 Invalid priority specified (must be 1..225)
130 Could not open program file
131 File System error while trying to read program file
132 Invalid program file (incorrect format)
133 Could not get a stack segment for new process
134 Could not get a syslocal segment for new process
135 Could not get sysglobal space for new process

136 Could not set up communication channel for new process
138 Error accessing program file while loading
141 Error accessing a library file while loading program
142 Can't run protected file on this machine
143 Program uses an intrinsic unit not found in the Intrinsic Library
144 Program uses an intrinsic unit whose name/type does not agree with the Intrinsic Library
145 Program uses a shared segment not found in the Intrinsic Library
146 Program uses a shared segment whose name does not agree with te Intrinsic Library
147 No space in syslocal for program file descriptor during process creation
148 No space in the shared IU data segment for the program's shared IU globals
190 No space in syslocal for program file description during List_LibFiles operation
191 Could not open program file
192 Error trying to read program file
193 Can't read protected program file
194 Invalid program file (incorrect format)
195 Program uses a shared segment not found in the Intrinsic Library
196 Program uses a shared segment whose name does not agree with the Intrinsic Library
198 Disk I/O error trying to read the intrinsic unit directory
199 Specified library file number does not exist in the Intrinsic Library
201 No such exception name declared
202 No space left in the system data area for declare_execp_hdl or signal_excep
203 Null name specified as exception name
302 Invalid ldsn
303 No data segment bound to the ldsn
304 Data segment already bound to the ldsn
306 Data segment too large
307 Input data segment path name is invalid
308 Data segment already exists
309 Insufficient disk space for data segment
310 An invalid size has been specified
311 Insufficient system resources
312 Unexpected file system error
313 Data segment not found
314 Invalid address passed to Info_Address
315 Operation may cause a data lockout
317 Disk error while trying to swap in data segment

401 Invalid event channel name passed to make_event_chn
402 No space left in system global data area for open_event_chn
403 No space left in system local data area for open_event_chn
404 Non-block structured device specified in pathname
405 Catalog is full in Make_Event_Chn or Open_Event_Chn
406 No such event channel exists in Kill_Event_Chn
410 Attempt to open a local event channel to send
411 Attempt to open event channel to receive when event channel has a receiver
413 Unexpected file system error in Open_Event_Chn
416 Cannot get enough disk space for event channel in Open_Event_Chn
417 Unexpected file system error in Close_Event_Chn
420 Attempt to wait on a channel that the calling process did not open
421 Wait_Event_Chn returns empty because sender process could not complete
422 Attempt to call wait_event_chn on an empty event-call channel
423 Cannot find corresponding event channel after being blocked
424 Amount of data returned while reading from event channel not of expected size
425 Event channel empty after being unblocked, Wait_Event_Chn
426 Bad request pointer error returned in Wait_Event_Chn
427 Wait_List has illegal length specified
428 Receiver unblocked because last sender closed
429 Unexpected file system error in Wait_Event_Chn
430 Attempt to send to a channel which the calling process does not have open
431 Amount of data transferred while writing to event channel not of expected size
432 Sender unblocked because receiver closed in Send_Event_Chn
433 Unexpected file system error in Send_Event_Chn
440 Unexpected file system error in Make_Event_Chn
441 Event channel already exists in Make_Event_Chn
445 Unexpected file system error in Kill_Event_Chn
450 Unexpected file system error in Flush_Event_Chn
530 Size of stack expansion request exceeds limit specified for program
531 Can't perform explicit stack expansion due to potential data space lock out
532 Insufficient disk space for explicit stack expansion
600 Attempt to perform I/O operation on non I/O request
602 No more alarms available during driver initialization
605 Call to non-configured device driver
606 Can't find sector on floppy diskette (disk unformatted)
608 Illegal length or disk address for transfer
609 Call to non-configured device driver

610 No more room in Sysglobal for I/O request
613 Unpermitted direct access to spare track with sparing enabled on floppy drive
614 No disk present in drive
615 Wrong call version to floppy drive
616 Unpermitted floppy drive function
617 Checksum error on floppy diskette
618 Can't format, or write-protected, or error unclamping floppy diskette
619 No more room in Sysglobal for I/O request
623 Illegal device control parameters to floppy drive
625 Scavenger indicated data is bad
630 The time passed to delay_time, convert_time, or send_event_chn has invalid year
631 Illegal Timeout request parameter
632 No memory available to initialize clock
634 Illegal Timed event id of -1
635 Process got unblocked prematurely due to process termination
636 Timer request did not complete successfully
638 Time passed to delay_time or send_event_chn more than 23 days from current time
639 Illegal date passed to Set_Time, or illegal date from system clock in Get_Time
640 RS-232 driver called with wrong version number
641 RS-232 read or write initiated with illegal parameter
642 Unimplemented or unsupported RS-232 driver function
646 No memory available to initialize RS-232
647 Unexpected RS-232 timer interrupt
648 Unpermitted RS-232 initialization, or disconnect detected
649 Illegal device control parameters to RS-232
652 N-port driver not initialized prior to Profile
653 No room in sysglobal to initialize Profile
654 Hard error status returned from drive
655 Wrong call version to Profile
656 Unpermitted Profile function
657 Illegal device control parameter to profile
658 Premature end of file when reading from driver
659 Corrupt file system header chain found in driver
660 Cable disconnected
662 Parity error while sending command or writing data to Profile
663 Checksum error or CRC error or parity error in data read
666 Timeout
670 Bad command response from drive
671 Illegal length specified (must = 1 on input)
672 Unimplemented console driver function

671 Illegal length specified (must = 1 on input)
672 Unimplemented console driver function
673 No memory available to initialize console
674 Console driver called with wrong version number
675 Illegal device control
680 Wrong call version to serial driver
682 Unpermitted serial driver function
683 No room in sysglobal to initialize serial driver
685 Eject not allowed this device
686 No room in sysglobal to initialize n-port card driver
687 Unpermitted n-port card driver function
688 Wrong call version to n-port card driver
690 Wrong call version to parallel printer
691 Illegal parallel printer parameters
692 N-port card not initialized prior to parallel printer
693 No room in sysglobal to initialize parallel printer
694 Unimplemented parallel printer function
695 Illegal device control parameters (parallel printer)
696 Printer out of paper
698 Printer offline
699 No response from printer
700 Mismatch between loader version number and operating system version number
701 OS exhausted its internal space during startup
702 Cannot make system process
703 Cannot kill pseudo-outer process
704 Cannot create driver
706 Cannot initialize floppy disk driver
707 Cannot initialize the file system volume
708 Hard disk mount table unreadable
709 Cannot map screen data
710 Too many slot-based devices
724 The boot tracks don't know the right file system version
725 Either damaged file system or damaged contents
726 Boot device read failed
727 The OS will not fit into the available memory
728 SYSTEM.OS is missing
729 SYSTEM.CONFIG is corrupt
730 SYSTEM.OS is corrupt
731 SYSTEM.DEBUG or SYSTEM.DEBUG2 is corrupt
732 SYSTEM.LLD is corrupt
733 Loader range error
734 Wrong driver is found. For instance, storing a Twiggy loader on a Profile
735 SYSTEM.LLD is missing

736 SYSTEM.UNPACK is missing
737 Unpack of SYSTEM.OS with SYSTEM.UNPACK failed
801 IoResult <> 0 on I/O using the Monitor
802 Asynchronous I/O request not completed successfully
803 Bad combination of mode parameters
806 Page specified is out of range
809 Invalid arguments (page, address, offset, or count)
810 The requested page could not be read in
816 Not enough sysglobal space for file system buffers
819 Bad device number
820 No space in sysglobal for asynchronous request list
821 Already initialized I/O for this device
822 Bad device number
825 Error in parameter values (Allocate)
826 No more room to allocate pages on device
828 Error in parameter values (Deallocate)
829 Partial deallocation only (ran into unallocated region)
835 Invalid s-file number
837 Unallocated s-file or I/O error
838 Map overflow: s-file too large
839 Attempt to compact file past PEOF
841 Unallocated s-file or I/O error
843 Requested exact fit, but one couldn't be provided
847 Requested transfer count is <= 0
848 End-of-file encountered
849 Invalid page or offset value in parameter list
852 Bad unit number (FlushFS)
854 No free slots in s-list directory (too many s-files)
855 No available disk space for file hints
856 Device not mounted
857 Empty, locked, or invalid s-file
861 Relative page is beyond PEOF (bad parameter value)
864 No sysglobal space for volume bitmap
866 Wrong FS version or not a valid Lisa FS volume
867 Bad unit number (Real_Mount, Real_Unmount)
868 Bad unit number (Def_Mount, Def_Unmount)
869 Unit already mounted (mount)/no unit mounted (unmount)
870 No sysglobal space for DCB or MDDF (mount)
871 Parameter not a valid s-file ID
872 No sysglobal space for s-file control block
873 Specified file is already open for private access
874 Device not mounted
875 Invalid s-file ID or s-file control block
879 Attempt to postion past LEOF
881 Attempt to read empty file

882 No space on volume for new data page of file
883 Attempt to read past LEOF
884 Not first auto-allocation, but file was empty
885 Could not update filesize hints after a write
886 No syslocal space for I/O request list
887 Catalog pointer does not indicate a catalog (bad parameter)
888 Entry not found in catalog
890 Entry by that name already exists
891 Catalog is full or is damaged
892 Illegal name for an entry
894 Entry not found, or catalog is damaged
895 Invalid entry name
896 Safety switch is on--cannot kill entry
897 Invalid bootdev value
899 Attempt to allocate a pipe
900 Invalid page count or FCB pointer argument
901 Could not satisfy allocation request
921 Pathname invalid or no such device (Make_File)
922 Invalid label size (Make_File)
926 Pathname invalid or no such device (Make_Pipe)
927 Invalid label size (Make_Pipe)
941 Pathname invalid or no such device (Kill_Object)
944 Object is not a file (Unkill_File)
945 File is not in the killed state (Unkill_File)
946 Pathname invalid or no such device (Open)
947 Not enough space in syslocal for file system refdb
948 Entry not found in specified catalog (Open)
949 Private access not allowed if file already open shared
950 Pipe already in use, requested access not possible or dwrite not allowed
951 File is already opened in private mode (Open)
952 Bad refnum (Close_Object)
954 Bad refnum (Read_data)
955 Read access not allowed to specified object
956 Attempt to position FMARK past LEOF not allowed
957 Negative request count is illegal (read_data)
958 Non-sequential access is not allowed (read_data)
959 System resources exhausted
960 Error writing to pipe while an unsatisfied read was pending
961 Bad refnum (write_data)
962 No WRITE or APPEND access allowed
963 Attempt to position FMARK too far past LEOF
964 Append access not allowed in absolute mode
965 Append access not allowed in relative mode
966 Internal inconsistency of FMARK and LEOF (warning)

967 Non-sequential access is not allowed (write_data)
968 Bad refnum (Flush)
971 Pathname invalid or no such device (Lookup)
972 Entry not found in specified catalog
974 Bad refnum (Info)
977 Bad refnum (Allocate)
978 Page count is non-positive (Allocate)
979 Not a block structured device (Allocate)
981 Bad refnum (Truncate)
982 No space has been allocated for specified file
983 Not a block structured device (Truncate)
985 Bad refnum (Compact)
986 No space has been allocated for specified file
987 Not a block structured device (Compact)
988 Bad refnum (Flush_Pipe)
989 Caller is not a reader of the pipe
990 Not a block structured device (Flush_Pipe)
994 Invalid refnum (Set_File_Info)
995 Not a block-structured device (Set_File_Info)
999 Asynchronous read was unblocked before it was satisfied
1021 Pathname invalid or no such entry (Rename_Entry)
1022 No such entry found (Rename_Entry)
1023 Invalid newname, check for '-' in string (Rename_Entry)
1024 New name already exists in catalog (Rename_Entry)
1031 Pathname invalid or no such entry (Read_Label)
1032 Invalid transfer count (Read_Label)
1033 No such entry found (Read_Label)
1041 Pathname invalid or no such entry (Write_Label)
1042 Invalid transfer count (Write_Label)
1043 No such entry found (Write_Label)
1051 No device or volume by that name (Mount)
1052 A volume is already mounted on device
1053 Attempt to mount temporarily unmounted boot volume just unmounted from this Lisa
1054 The bad block directory of the diskette is invalid
1061 No device or volume by that name (Unmount)
1062 No volume is mounted on device
1071 Not a valid or mounted volume for working directory
1091 Pathname invalid or no such entry (Set_Safety)
1092 No such entry found (Set_Safety)
1101 Invalid device name (DEVICE_CONTROL)
1121 Invalid device, not mounted, or catalog is damaged (Reset_catalog)
1128 Invalid pathname, device, or volume not mounted (Get_dev_name)
1130 File is protected; cannot open due to protection violation
1131 No device or volume by that name

1132 No volume is mounted on that device
1133 No more open files in the file list of that device
1134 Cannot find space in sysglobal for open file list
1135 Cannot find the open file entry to modify
1136 Boot volume not mounted
1137 Boot volume already unmounted
1138 Caller cannot have higher priority than system processes when calling ubd
1141 Boot volume was not unmounted when calling rbd
1142 Some other volume still mounted on the boot device when calling rbd
1143 No sysglobal space for MDDF to do rbd
1144 Attempt to remount volume which is not the temporarily unmounted boot volume
1145 No sysglobal space for bit map to do rbd
1158 Track-by-track copy buffer is too small
1159 Shutdown requested while boot volume was unmounted
1160 Destination device too small for track-by-track copy
1161 Invalid final shutdown mode
1162 Power is already off
1163 Illegal command
1164 Device is not a Twiggy device
1165 No volume is mounted on the device
1166 A valid volume is already mounted on the device
1167 The Device is not blockstructured
1168 Device name is invalid
1169 Could not default mount volume before initialization
1170 Could not mount volume after initialization
1171 '-' is not allowed in a volume name
1172 No space available to initialize a bitmap for the volume
1176 Cannot read from a pipe more than half of the allocated physical size
1177 Cannot cancel a read request for a pipe
1178 Process waiting for pipe data got unblocked because last pipe writer closed it
1180 Cannot write to a pipe more than half of the allocated physical size
1181 No system space left for request block for pipe
1182 Writer process to a pipe got unblocked before the request was satisfied
1183 Cannot cancel a write request for a pipe
1184 Process waiting for pipe space got unblocked because the reader closed the pipe
1186 Cannot allocate space to a pipe while it has data wrapped around
1188 Cannot compact a pipe while it has data wrapped around
1190 Attempt to access a page that is not allocated to the pipe

1191 Bad parameter (FileIO)
1193 Premature end of file encountered (FileIO)
1196 Something is still open on device--cannot unmount
1197 Volume is not formatted or cannot be read
1198 Negative request count is illegal
1199 Function or procedure is not yet implemented
1200 Illegal volume parameter
1201 Blank file parameter
1202 Error writing destination file
1203 Invalid UCSD directory
1204 File not found
1210 Boot track program not executable
1211 Boot track program too big
1212 Error reading boot track program
1213 Error writing boot track program
1214 Boot track program file not found
1215 Can't write boot tracks on that device
1216 Couldn't create/close internal buffer
1217 Boot track program has too many code segments
1218 Couldn't find configuration information entry
1219 Couldn't get enough working space
1220 Premature EOF in boot track program
1221 Position out of range
1222 No device at that position
1225 Scavenger has detected an internal inconsistency symptomatic of a software bug
1226 Invalid device name
1227 Device is not block structured
1228 Illegal attempt to scavenge the boot volume
1229 Cannot read consistently from the volume
1230 Cannot write consistently to the volume
1231 Cannot allocate space (Heap segment)
1232 Cannot allocate space (Map segment)
1233 Cannot allocate space (SFDB segment)
1237 Error rebuilding the volume root directory
1240 Illegal attempt to Scavenge a non OS formatted volume
1296 Bad string argument has been passed
1297 Entry name for the object is invalid (on the volume)
1298 S-list entry for the object is invalid (on the volume)
1807 No disk in floppy drive
1820 Write protect error on floppy drive
1822 Unable to clamp floppy drive
1824 Floppy drive write error
1882 Bad response from Profile
1885 Profile timeout error

1998 Invalid parameter address
1999 Bad refnum

OPERATING SYSTEM ERROR CODES

The error codes listed below are generated only when a non-recoverable error occurs while in operating system code. The errors are listed by functional modules of the OS.

SYSTEM ERRORS FOR THE ASYNCHRONOUS CONTROL UNIT

10050 Request block is not chained to a pcb (unblk_req)
10051 bld_req is called with interrupts off

SYSTEM ERRORS IN PROCESS MANAGEMENT

10701 No space during StartUp for system segment setup list or global process list head (Init_GPList and AllocSys_Segs)
10100 An error was returned from SetUp_Directory (Get_UnitDir_Entry and Change_Directory)
10101 Couldn't find unit BlkIO or segment PasLib during Change_Directory
10102 Error>130 trying to create shell (Root)
10103 sem_count>1 (Init_Sem)
10104 Couldn't GetSpace in syslocal for an ObjDescriptor (InitObjFile)
10105 Couldn't GetSpace in IU shared data segment after 20 tries (Get_Shared_ptr)
10197 Automatic stack expansion fault occurred in system code (Check_Stack)
10198 Need_mem set for current process while scheduling is disabled (SimpleScheduler)
10199 Attempt to block for reson other than I/O while scheduling isdisabled (SimpleScheduler)

SYSTEMS ERRORS IN EXCEPTION MANAGEMENT

10200 No space left in system data area in Hard_excep
10201 Hardware exception occurred while in system code
10202 No space left from sigl_excep call in hard_excep
10203 No space left from sigl_excep call in nmi_excep
10204 Error from info_event_chn called in get_evt_num
10205 Error from wait_event_chn called in excep_prolog
10207 No system data space in excep_setup
10208 No space left from sigl_excep call in rangeerror

10212 Error in term_def_hdl from enable_excep
10213 Error in force_term_excep, no space in enq_ex_data

SYSTEM ERRORS IN EVENT CHANNEL MANAGEMENT

10401 Error from close_event_chn in ec_cleanup
10402 Actual returned from write_data for timer event is not correct

SYSTEM ERRORS IN MEMORY MANAGEMENT

10579 Unable to swap in OS code segment
10580 Unagle to get space in Bld_Seg
10581 Unable to get space in MM_Setup
10582 Unable to get space in Freeze_Seg
10590 Fatal parity error
10593 Unable to move memory manager segment during startup
10594 Unable to swap in a segment during startup
10595 Unable to get space in Extend_MMlist
10596 Trying to alter size of segment that is not data or stack (Alt_DS_Size)
10597 Trying to allocate space to an allocted segment (Alloc_Mem)
10598 Attempting to allocate a non-free memory region (Take_Free)

SYSTEM ERRORS IN DRIVER CODE

10605 Interrupt from non-configured device
10609 Interrupt from non-configured device
10611 Spurious interrupt from Twiggy drive #2
10612 Spurious interrupt from Twiggy drive #1 ***Duplicate sys error ****
10633 Got timeout interrupt with no requests to timeout
10637 No more "alarms" available for timeout request
10651 Spurious Profile interrupt
10695 Spurious Parallel printer interrupt
10695 Spurious Parallel printer alarm interrupt

SYSTEM ERRORS IN TIME MANAGEMENT

10600 Error from make_pipe to make timer pipe
10601 Error from kill_object of the existing timer pipe
10602 Error from second make_pipe to make timer pipe
10603 Error from open to open timer pipe
10604 No syslocal space for head of timer list

10610 Error from info about timer pipe
10612 No syslocal space for timer list element
10613 Error from read_data of timer pipe
10614 Actual returned from read_data is not the same as requested from timer pipe
10615 Error from open of the receiver's event channel
10616 Error from write_event to the receiver's event channel
10617 Error from close_event_chn on the receiver's pipe

Appendix E
FS__INFO FIELDS

DEVICE_T, VOLUME_T:

	backup_volid	ID of the volume of which this volume is a copy.
	blocksize	Number of bytes in a block on this device.
*	blockstructured	Flag set if this device is block-structured.
	boot_code	Reserved.
	boot_environ	Reserved.
	clustersize	Reserved.
	copy	Reserved.
	copy_flag	Flag set if this volume is a copy.
	copy_thread	Count of copy operations involving this volume.
	datasize	Number of data bytes in a page on this volume.
*	devt	Device type.
*	dir_path	Pathname of the volume/device.
	DTCC	Date/time volume was created if it is a copy.
	DTVB	Date/time volume was last backed-up.
	DTVC	Date/time volume was created.
	DTVS	Date/time volume was last scavenged.
	filecount	Count of files on this volume.
	freecount	Count of free pages on this volume.
	fs_overhead	Number of pages on this volume required to store file system data structures.
	fs_size	Number of pages on this volume.
	fsversion	Version number of the file system under which this volume was initialized.
*	iochannel	Number of the expansion card channel through which this device is accessed.
	label_size	Size in bytes of the user-defined labels associated with objects on this volume.
$	lockeddev	Reserved.
	machine_ID	Machine on which this volume was initialized.
	master	Reserved.
	master_copy_ID	Reserved.
*	mounted	Flag set if a volume is mounted.
$	mount_pending	Reserved.
*	name	Name of this volume/device.
$	opencount	Count of objects open on this volume/device.

overmount_stamp	Reserved.
password	Password of this volume.
$ privatedev	Reserved.
privileged	Reserved.
$ remote	Reserved.
result_scavenge	Reserved.
scavenge_flag	Flag set by the Scavenger if it has altered this volume in some way.
* slot_no	Number of the expansion slot holding the card through which this device is accessed.
$ unmount_pending	Reserved.
volid	Unique identifier for this volume.
vol_left_mounted	Flag set if this volume was mounted during a system crash.
volname	Volume name.
volnum	Volume number.
vol_size	Total number of blocks in the file system volume and boot area on this device.
write_protected	Reserved.

* defined for mounted or unmounted devices
$ defined for mounted devices only
(all other fields are defined for mounted block-structured devices only)

OBJECT_T:

acmode	Set of access modes associated with this refnum.
dir_path	Pathname of the directory containing this object.
DTA	Date/time object was last accessed.
DTB	Date/time object was last backed-up.
DTC	Date/time object was created.
DTM	Date/time object was last modified.
DTS	Date/time object was last scavenged.
eof	Flag set if end-of-file has been encountered on this object (through the given refnum).
etype	Directory entry type.
file_closed_by_OS	Flag set if this object was closed by the operating system.
file_left_open	Flag set if this object was open during a system crash.
file_scavenged	Flag set by the Scavenger if this object has been altered in some way.

fmark	Absolute byte at which the file mark points.
fs_overhead	Number of pages used by the file system to store control information about this object.
ftype	Object type.
fuid	Unique identifier for this object.
kswitch	Flag set when the object is killed.
locked	Reserved.
lpsize	Number of data bytes on a page.
machine_ID	Machine on which this object may be opened.
master_file	Flag set if this object is a master.
name	Entry name of this object.
nreaders	Number of processes with this object open for reading.
nwriters	Number of processes with this object open for writing.
nusers	Number of processes with this object open.
private	Flag set if this object is open for private access.
protected	Flag set if this object is protected.
psize	Physical size of this object in bytes.
refnum	Reference number for this object (argument to INFO).
result_scavenge	Reserved.
safety_on	Value of the safety switch for this object.
size	Number of data bytes in this object (LEOF).
system_type	Reserved.
user_type	User-defined type field for this object.
user_subtype	User-defined subtype field for this object.

----------A----------
accessing devices: 1.3
ACTIVATE_PROCESS: 3.8
ALLOCATE: 2.10
ALTCONSOLE: 2.1
attribute: 1.3

----------B----------
binding: 4.1
BIND_DATASEG: 4.7
BITBKT: 2.1
blocked process: 1.4, 3

----------C----------
CARDS_EQUIPPED: 6.1
catalog: 2, 2.1, 2.10
clock: 5.6
- system calls: 5.9
CLOSE_DATASEG: 4.7
CLOSE_EVENT_CHN: 5.8
CLOSE_OBJECT: 2.10
code segment: 4.5
communication between processes: 1.7
COMPACT: 2.10
configuration: 6
- system calls: 6.1
controlling a device: 2.10
- a process: 3.4
CONVERT_TIME: 5.9
copying a file: 2.10
creating a data segment (MAKE_DATASEG): 4.7
- a process (MAKE_PROCESS): 3.3, 3.8
- an event channel (MAKE_EVENT_CHN): 5.8
- an object (MAKE_FILE, MAKE_PIPE): 2.10

----------D----------
data segment, local: 4.1
- private: 4.1, 4.4
- shared: 1.7, 4.1, 4.3
- swapping: 4.6
dccode: 2.10
dcdata: 2.10
dctype: 2.10
dcversion: 2.10

DECLARE_EXCEP_HDL: 5.7
DELAY_TIME: 5.9
device: 2.3
- names (predefined): 2.1
DEVICE_CONTROL: 2.10
directory: 2
DISABLE_EXCEP: 5.7
disk hard error codes: 2.10

----------E----------
enabled exception: 5.1
ENABLE_EXCEP: 5.7
end of file, logical: 2, 2.7
- physical: 2, 2.7
error messages: D
event: 1.6, 5, 5.4
- channel: 1.7, 5.5
- types: C
event mangagement system calls: 5.8
exception: 1.6, 5
- enabled: 5.1
- handlers: 5.3
- ignored: 5.1
- names: B
- queued: 5.1
exception management system calls: 5.7

----------F----------
father process: 1.4
file: 2
- access: 2.8
- label: 2, 2.6
- marker: 2, 2.7
- name: 2.1
- private: 2.8
- shared: 1.7, 2.8
file system: 1.3, 2
- calls: 2.10
FLUSH: 2.10
FLUSH_DATASEG: 4.7
FLUSH_EVENT_CHN: 5.8
FLUSH_EXCEP: 5.7
FS_INFO fields: E

----------G----------

GET_CONFIG_NAME: 6.1
GET_NEXT_ENTRY: 2.10
GET_TIME: 5.9
GET_WORKING_DIR: 2.10
global access to files: 2.8

----------H----------
hard error: 2.10
hierarchy of processes: 3.2

---------I,J---------
ignored exception: 5.1
INFO: 2.10
INFO_ADDRESS: 4.7
INFO_DATASEG: 4.7
INFO_EVENT_CHN: 5.8
INFO_EXCE: 5.7
INFO_LDSN: 4.7
INFO_PROCESS: 3.8
input & output: 2
interprocess communication: 1.7
I/O: 2

----------K----------
KILL_DATASEG: 4.7
KILL_EVENT_CHN: 5.8
KILL_OBJECT: 2.10
KILL_PROCESS: 3.8

----------L----------
label: 1.3
LDSN: 4.2
LEOF: 2, 2.7
local data segment: 4.1
local data segment number: 4.2
logical end of file: 2, 2.7
LOOK_UPF: 2.10
LOWER: 2.1

---------M,N---------
MAINCONSOLE: 2.1
MAKE_DATASEG: 4.7
MAKE_EVENT_CHN: 5.8
MAKE_FILE: 2.10
MAKE_PIPE: 2.10

MAKE_PROCESS: 3.8
MDDF: 2.4
medium descriptor data file: 2.4
memory management: 1.5, 4
- system calls: 4.7
MEM_INFO: 4.7
MMU: 4
MOUNT: 2.10
mounting a device: 1.3
MY_ID: 3.8
naming a device: 1.3
naming a file: 1.3

----------O----------
OPEN: 2.10
OPEN_DATASEG: 4.7
OPEN_EVENT_CHN: 5.8
OS interface: A
OS_BOOT_VOL: 6.1

---------P,Q---------
page: 2.4
- descriptor: 2.4
parameter memory: 6
PARAPORT: 2.1
pathname: 1.3
PEOF: 2, 2.7
physical end of file: 2, 2.7
pipe: 1.7, 2.9
priority of devices: 2.3
private data segment: 4.1, 4.4
private file: 2.8
process: 1.4, 3
- blocked: 1.4, 3
- control: 3.4
- creation: 3.3
- father: 1.4
- hierarchy: 3.2
- ready: 1.4, 3
- running: 1.4, 3
- scheduling: 3.5
- son: 1.4, 3
- structure: 3.1
- terminated: 1.4, 3
process system calls: 3.8

queued exception: 5.1

----------R----------
ready process: 1.4, 3
READ_DATA: 2.10
READ_LABEL: 2.10
refnum: 2.8
RENAME_ENTRY: 2.10
RESET_CATALOG: 2.10
RS232A: 2.1
RS232B: 2.1
running process: 1.4, 3

----------S----------
scheduler: 3
scheduling processes: 3.5
SEND_EVENT_CHN: 5.8
SETACCESS_DATASEG: 4.7
SETPRIORITY_PROCESS: 3.8
SET_FILE_INFO: 2.10
SET_LOCAL_TIME_DIFF: 5.9
SET_SAFETY: 2.10
SET_WORKING_DIR: 2.10
shared data segment: 1.7, 4.1, 4.3
shared file: 1.7, 2.8
SIGNAL_EXCEP: 5.7
SIZE_DATASEG: 4.7
SLOTxCHANy: 2.1
soft error: 2.10
son process: 1.4
storage device: 2.3
structure of processes: 3.1
SUSPEND_PROCESS: 3.8
swapping: 4.6
system calls, clock: 5.9
- configuration: 6.1
- event management: 5.8
- exception management: 5.7
- memory management: 4.7
- process: 3.8
system clock: 5.6
system defined exceptions: 5.2
SYS_OVERFLOW: B
SYS_SON_TERM: C
SYS_TERMINATE: B

SYS_VALUE_OOB: B
SYS_ZERO_DIV: B

----------T----------
terminated process: 1.4, 3.6
TERMINATE_PROCESS: 3.8
TRUNCATE: 2.10

----------U----------
UNBIND_DATASEG: 4.7
UNKILL_FILE: 2.10
UNMOUNT: 2.10
UPPER: 2.1

----------V----------
volume catalog: 2.5
volume name: 1.3

-------W,X,Y,Z-------
WAIT_EVENT_CHN: 5.8
working directory: 2.2
WRITE_DATA: 2.10
WRITE_LABEL: 2.10
YIELD_CPU: 3.8

THIS MANUAL was produced using LisaWrite, LisaDraw, and LisaList.

ALL PRINTING was done with an Apple Dot-Matrix Printer.

Lisa™

...we use it ourselves.