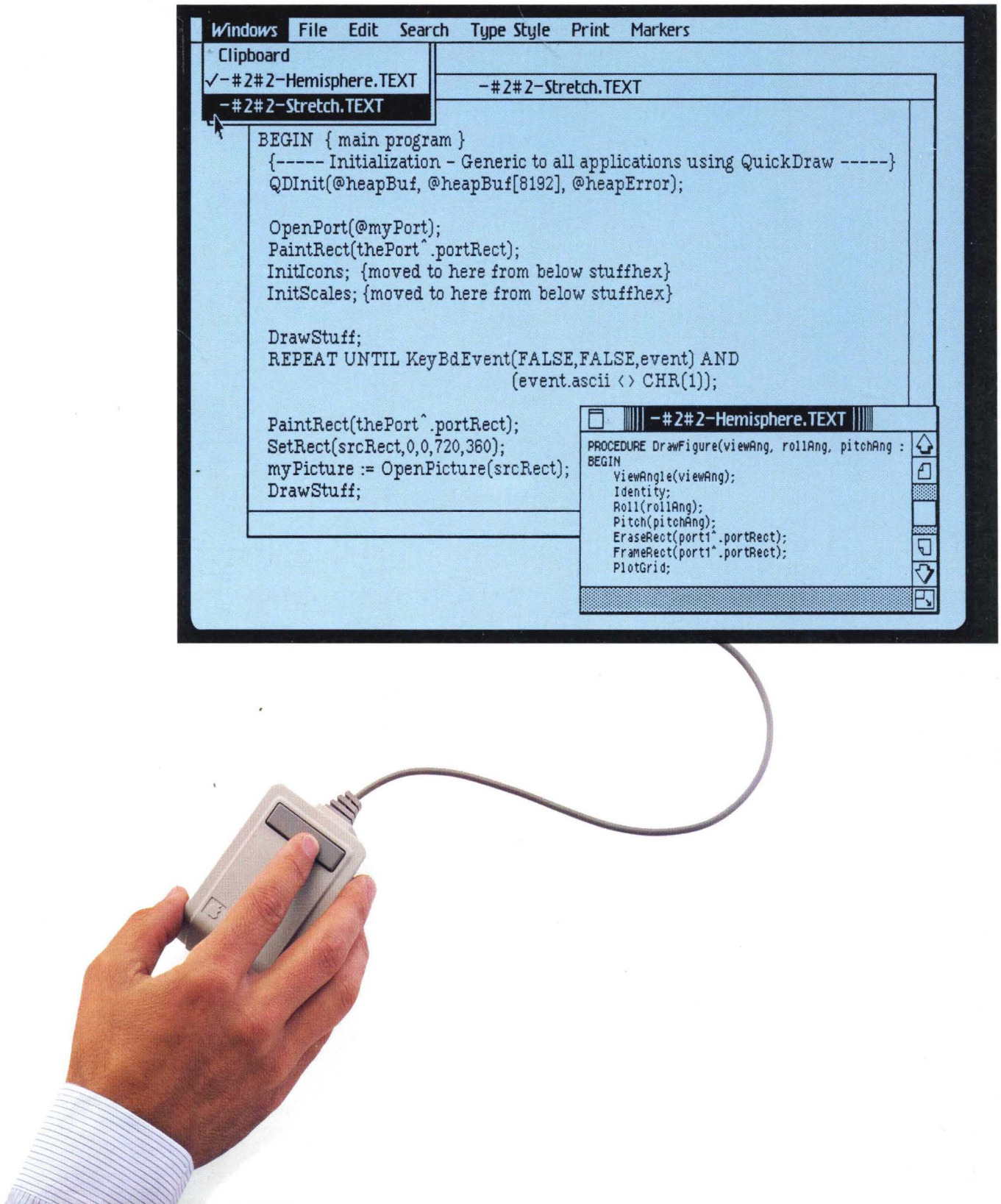
Lisa Systems Software



Lisa Pascal 3.0 Systems Software

Licensing Requirements for Software Developers

Apple has a low-cost licensing program, which permits developers of software for the Lisa to incorporate Apple-developed libraries and object codes into their products. Both in-house and external distribution require a license. Before distributing any products that incorporate Apple software, please contact Software Licensing at the address below for both licensing and technical information.

20525 Mariani Ave.
Cupertino, CA 95014
(408) 996-1010

Simultaneously published in the U.S.A. and Canada.

Reorder Apple Product #620-6149-B.

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed, or in the manuals distributed with the software, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THE PRODUCT.

Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

IN NO EVENT WILL APPLE BE HELD LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

What's Inside

This binder contains seven documents about the Lisa™ system software for programmers' reference. The manuals are, in order:

- *Operating System Reference Manual for the Lisa.*
- *The OEMSyscall Unit.*
- *The Standard Apple Numeric Environment.*
- *The 68000 Assembly-Language SANE.*
- *The StdUnit.*
- *The ProgComm Unit.*
- *The QuickPort Programmer's Guide.*

In addition, elsewhere in this package of books and media, there is a copy of Motorola's *M68000 16/32 Bit Microprocessor Programmer's Reference Manual.*

Operating System Reference Manual for the Lisa

Contents

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 that control process synchronization. It also describes the use of the system clock.

Chapter 6 describes the calls you can use to find out 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-reserved exception names.

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

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

Appendix E contains a description of the information you can obtain from the Operating System about files and devices.

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 capitalized in this manual, although Pascal does not distinguish between lower and upper case characters. *Italics* indicate a new term whose explanation follows.

Future Releases

A few features of the Lisa Operating System will be changed in future releases:

- Pipes will not be supported.
- Timed events will not be supported.
- Configuration System Calls will be changed.

If you want your software to be upward-compatible, please take these changes into consideration. More information is provided in the appropriate sections of the manual.

Chapter 1
Introduction

Introduction

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

1.1 The Main Functions

This chapter describes the four main functional areas of the OS: 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 consists of 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 management 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.

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 **writeln** is supplied in IOSPASLIB, the Pascal run-time support routines library. This code, which is added to the program when the program is linked, 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. For a list of error codes and their meaning, see Appendix D.

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, you specify its file name. To access an object in a different directory, you specify its complete pathname.

Before a device can be accessed, it must be mounted. Devices can be mounted using the Preferences tool or by using the **MOUNT** call. See Chapter 2 for an explanation of this call and other File System calls. 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.

A device can be accessed if it is specified in the configuration list created by the Preferences tool, is physically connected to the Lisa, and is mounted. There are some operations that can be performed on unmounted devices. Two examples are **DEVICE_CONTROL** calls and scavenging. 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 act 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 - control 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 called *attributes*, such as the length and creation date of a file. Calls are available to access the attributes of any File System object. In addition to its system-defined attributes, an object 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 that provide interprocess 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 the data associated with it. 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. The blocked state is explained later in this section.

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 for an illustration of a process tree.

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 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 memory segments. Each segment can contain up to 128 Kbytes. Memory 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 permanently 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 either by the system or by an executing program. System exceptions are generated by various sorts of errors such as divide by zero, illegal instruction, and 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 can be 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 expects 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 an event-call 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: 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. (Note that pipes will not be supported in future releases of the Operating System.) The pipe mechanism provides for the needed synchronization; 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 that event channels 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 **Syscall** unit, found in Appendix A. 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, which provides program development tools such as editing and debugging facilities.

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 **Syscall** unit and then calls the routines needed.

Chapter 2
The File System

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 can also be 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. However, on block-structured devices, programs can make I/O requests in whole-block increments in order to improve 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
```

refers to a file named FORMAT.TEXT 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: 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 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 non-printing characters and blank spaces. However, use of ASCII 13, 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 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; 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 which do not contain any file data can be allocated to the file. There are, therefore, two measures of the end of the 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 called the *file marker* which 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 automatically 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 are written. The file marker cannot be positioned past LEOF except by a write

operation that appends data to a file; in this case the file marker is positioned one byte past LEOF.

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 bytestream interface. As far as an application 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 perform only sequential operations. Programs can sometimes improve performance, however, 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 to 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 by more than one process concurrently. 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 the 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 ensure 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 *global access mode*, 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 refnums 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.

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 similar to the other objects in the File System -- they are named according to the same rules, and they can have labels.

NOTE

Pipes will not be supported in future releases of the Operating System. Do not use the pipe mechanism if you want your software to be upward-compatible.

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 be accessed only 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 has the pipe 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 restrictions on some operations. 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 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.

2.10.1 MAKE_FILE and MAKE_PIPE File System Calls

```
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 pathname does not begin with a dash), the working directory is used. Label_Size specifies the initial size in bytes of the label. 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.

NOTE

Pipes will not be supported in future releases of the Operating System. Do not use the pipe mechanism if you want your software to be upward-compatible.

The MAKE_FILE example on the next page 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;
```

2.10.2 KILL_OBJECT File System Call

```
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 RETURN is pressed:

```
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;
```

2.10.3 UNKILL_FILE File System Call

```
UNKILL_FILE  (Var Ecode:Integer;
                  RefNum:Integer;
              Var Newname:e_name)
```

```
Ecode:   Error indicator
RefNum:  Refnum of the killed and open file
Newname: 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 Newname. Newname is not a full pathname: the resurrected file remains in the same directory.

2.10.4 RENAME_ENTRY File System Call

```
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** cannot 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 full pathname after renaming 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;
```

2.10.5 LOOKUP File System Call

```
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, all of 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 scavenged *)
            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 Modified  *)
            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, MDOFFile, 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;
```

2.10.6 INFO File System Call

```
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 that are open. The definition of the **Fs_Info** record is given under **LOOKUP** and in Appendix A.

2.10.7 SET_FILE_INFO File System Call

```
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 a 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_scavenged
file_closed by_OS
file_left_open
user_type
user_subtype

2.10.8 OPEN File System Call

```
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 the file
- **Dwrite** -- Allows you to write in the file (to replace 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 specify **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. To set up a private pipe, the reader process opens the pipe first, specifying **Dread** mode; the writer process then opens the pipe with **Append**, **Private** access mode.

2.10.9 CLOSE_OBJECT File System Call

```
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, and 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;
```

2.10.10 READ_DATA and WRITE_DATA File System Calls

```
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, **Offset** specifies a byte relative to the file marker. In sequential mode, **Offset** is ignored (assumed to be zero); transfers occur relative to the file marker. Sequential mode (which is a special case of relative mode) is the only access mode allowed 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 ensure that any newly allocated blocks are located next to each other, although they may not be 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 there are no

writers, the end-of-file indication (error 848) is returned in 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 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 By te=-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.
```

2.10.11 READ_LABEL and WRITE_LABEL File System Calls

```
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, 128 bytes.

2.10.12 DEVICE_CONTROL File System Call

```
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: currently 2
dcCode: control code for device driver
dcData: specific control or data parameters

2.10.12.1 Setting Device-Control Information

Before you use a device, you call **DEVICE_CONTROL** to set the device driver. Once you begin using the device, you call **DEVICE_CONTROL** as necessary.

Table 2-1 shows which groups of device-control functions must be set before using each type of device. Table 2-2 shows which characteristics are contained in each group. For example, you must set Group A for RS-232 input. As you see in Table 2-2, Group A indicates the type of parity used with the device. Each group requires a separate call to **DEVICE_CONTROL**, and you can set only 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 RS232 for input	RS232A or RS232B	A, C, D, E, F, G, L, M, N
Serial RS232 for output or printer	RS232A or RS232B	A, B, C, G, H, I, M, N
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 that 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 requires only 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, 8 bits of data	1	0	--	--
Odd parity, 7 bits of data	1	1	--	--
Even parity, 7 bits of data	1	3	--	--
8 bits of data plus ninth bit odd parity	1	5	--	--
No parity, input stripped to 7 bits	1	6	--	--
Group B, Output Handshake:				
None	11	--	--	--
DTR handshake	2	--	--	--
XON/XOFF handshake	3	--	--	--
delay after CR, LF	4	ms delay	--	--
Group C[1], Baud rate:				
	5	baud	--	--
Group D, Input waiting during **Read_Data**:				
wait for **Count** bytes	6	0	--	--
return whatever rec'd	6	1	--	--
Group E[2], Input handshake:				
no handshake	7	--	--	--
	9	-1	-1	32767
DTR handshake	7	--	--	--
XON/XOFF handshake	8	--	--	--
Group F[3], Input typeahead buffer:				
flush only	9	-1	-2	-2
flush and resize	9	bytes	-2	-2
flush, resize, and set threshold	9	bytes	low	hi

Table 2-2 (continued)

FUNCTION	.dccode	.dcdata[0]	.dcdata[1]	.dcdata[2]
Group G, Disconnect Detection:				
none	10	0	0	--
BREAK detected means disconnect	10	0	nonzero	--
Group H, Timeout on output (handshake interval):				
no timeout	12	0	--	--
timeout enabled	12	seconds	--	--
Group I, Automatic linefeed insertion:				
disabled	17	0	--	--
enabled	17	1	--	--
Group J[4], Disk errors (set to 1 to enable, to 0 to disable):				
enable sparing	21	sparing	rewrite	reread
Group K[5], Break command (never required, available only on serial RS232 devices):				
send break	13	millisecond duration	0	--
send break while lowering DTR	13	millisecond duration	1	--
Group L, Timeout on Input:				
No timeout	14	0	--	--
Timeout enabled	14	seconds	--	--
Group M, BREAK during Close_Object:				
enabled (default)	25	nonzero	--	--
disabled	25	0	--	--
Group N[6], Set Modem Timeouts (Int'l MODEM A driver only):				
Set timeouts	22	recovery	carrier	connect
Group P, Wait until modem connects (Int'l MODEM A driver only)				
Wait (returns with errnum=645 if no connect)	24	--	--	--

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.

2. In **Group E**, to specify no input handshake, first make the call with the device control code 7, then call again with the device control code 9, as shown.

3. *Low* and *Hi* under **Group F** set the low and high threshold in the typeahead input buffer. When Hi or more bytes are in the input buffer, XOFF is sent or DTR is dropped. When Low or fewer bytes remain in the typeahead buffer, XON is sent or DTR is reasserted. The size of the typeahead buffer (bytes) can be any value between 0 and 1024 bytes inclusive.

4. In **Group J**, enabling disk sparing lets the device driver to relocate blocks of data from areas of the disk that are found to be bad. Enabling disk rewrite allows 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*. Enabling disk reread tells the Operating System to read data after they are written to make certain that they were written correctly.

5. When sending a break command, as 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.

6. In **Group N**, *recovery* is the minimum number of milliseconds required by the modem between calls. *Carrier* is the number of milliseconds without carrier detect, before the driver disconnects from the line. *Connect* is the maximum number of seconds the driver will wait when **Group P** Device_Control is subsequently issued.

Table 2-3 gives a list of mnemonic constants that you can use in place of explicit numbers when setting **Dccode**. These mnemonics are provided in the SysCall unit for convenience.

Table 2-3
Dccode Mnemonics

Dccode	Mnemonic	Dccode	Mnemonic
1	dvParity	14	no mnemonic
2	dvOutDTR	15	dvErrStat
3	dvOutXON	16	dvGetEvent
4	dvOutDelay	17	dvAutoLF
5	dvBaud	18	no mnemonic
6	dvInWait	19	no mnemonic
7	dvInDTR	20	dvDiskStat
8	dvInXON	21	dvDiskSpare
9	dvTypeahd	22	no mnemonic
10	dvDiscon	23	no mnemonic
11	dvOutNoHS	24	no mnemonic
12	no mnemonic	25	no mnemonic
13	no mnemonic		

2.10.12.2 Obtaining Device-Control Information

To use **DEVICE_CONTROL** to find out about the current state of a particular device, simply give the pathname for the particular device along with a function code for the type of information you need. 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 applies only to some of the available devices. The request types and the returned information are described in Table 2-4.

Table 2-5 shows the error code provided in response to a **Dccode=15** information request. This code is given in **cparm.dcdata[0]**. The code, a long integer, is shown in Table 2-5; the bits and bytes are numbered from the right, counting from 0. The meaning assigned to the bit applies if the bit is set (equals 1).

Here is a program fragment that uses **DEVICE_CONTROL** to get information about the lower diskette drive.

```
VAR
   cparm: dctype;
   errnum: INTEGER;
   path: pathname;
BEGIN
   path:='-LOWER';
   cparm.dcversion:=2;    (* always set this value *)
```

```
    cparm.dccode := 20;
    DEVICE_CONTROL(errnum, path, cparm);
    WITH cparm DO
    WRITELN (dcdata[0], dcdata[1], dcdata[2], dcdata[3],
             dcdata[4], dcdata[5], dcdata[6])
END;
```

Table 2-4
Device Information

Dccode	Devices	Returned in Dcdata
15	ProFiles	**[0]** contains disk error status on last hardware error (see 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

Table 2-4 (continued)

Dccode	Devices	Returned in Dcdata
18	RS232, Modem A	Read and clear input error counters **[0]** contains count of framing errors **[1]** contains count of parity errors **[2]** contains count of overrun errors **[3]** is count of buffer overflow errors
19	RS232, Modem A	**[0]** returns last value passed in Group A, **Dcdata[0]** **[1]** returns last value passed in **dccode** for Group B, or negative value of 'ms delay' if 'delay after CR,LF' was selected **[2]** returns baud rate **[3]** upper 16 bits: returns last value from **dcdata[0]**, Group D lower 16 bits: returns last value from **dccode**, Group E **[4]** returns value from 'bytes' Group F **[5]** upper 16 bits: value from 'low', Group F lower 16 bits: value from 'hi', Group F **[6]** returns 'seconds' from group H **[7]** upper 16 bits: value from **dcdata[1]** Group G lower 16 bits: value from **dcdata[0]** Group I **[8]** returns value from **dcdata[0]**, Group L **[9]** returns number of characters waiting in driver's input buffer

Table 2-4 (continued)

Dccode	Devices	Returned in Dcdata
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 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 available spare 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
23	Modem A	**[0]** returns 'recovery', Group N **[1]** returns 'carrier', Group N **[2]** returns 'connect', Group N **[3]** returns: 0 = not connected 1 = connected

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 = Host's 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.

2.10.13 ALLOCATE File System Call

```
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 error 1186 in this case.

2.10.14 COMPACT File System Call

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

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

COMPACT changes the Physical End of File to deallocate 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 in the case of **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 error 1188 in this case.

2.10.15 TRUNCATE File System Call

```
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 changes only LEOF. COMPACT, on the other hand, changes only PEOF.

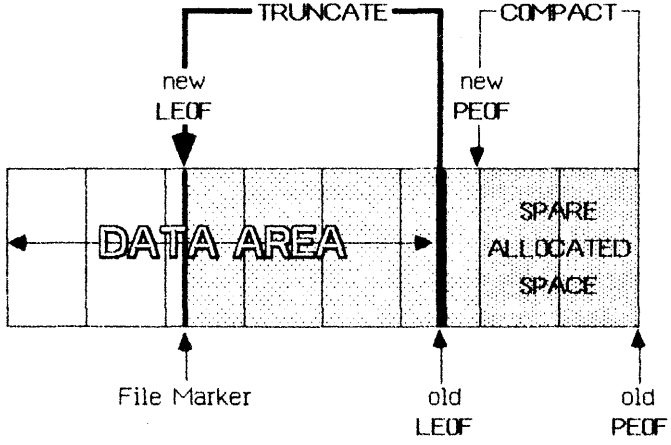


Figure 2-2
The Relationship of COMPACT and TRUNCATE

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

2.10.16 FLUSH File System Call

```
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 ensure that the File System is consistent.

2.10.17 SET_SAFETY File System Call

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

        Ecode:   Error indicator
        Path:    Name of object containing safety switch
        On_Off:  Set safety switch:
                     On  = true
                     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.

2.10.18 SET_WORKING_DIR and GET_WORKING_DIR File System Calls

```
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 working directory name 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;
```

2.10.19 RESET_CATALOG, RESET_SUBTREE, GET_NEXT_ENTRY, and LOOKUP_NEXT_ENTRY File System Calls

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

RESET_SUBTREE (var ecode : integer;
               var path : pathname)

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

LOOKUP_NEXT_ENTRY (var ecode : integer;
                   var prefix : e_name;
                   var InfoRec : Q_Info)
```

```
ecode:   Error indicator
path:    Name of the directory to be scanned
prefix:  Find names beginning with this substring
entry:   Name of the next object (with matching
         prefix) in the directory
```

These procedures are used to enumerate the objects contained in a directory. **RESET_CATALOG** instructs the file system that the directory named in **path** is to be scanned. **GET_NEXT_ENTRY** returns the name of the next object in the directory. Only names beginning with the substring **prefix** will be found. If prefix is the null string, then all names in the directory will be found. If there are no more objects in the directory, an end-of-file error (848) is returned. **RESET_SUBTREE** is equivalent to **RESET_CATALOG**, but indicates that the subtree rooted at the directory named in **path** is to be scanned. Subsequent calls to **GET_NEXT_ENTRY** will return names from the subtree according to a pre-order traversal. **LOOKUP_NEXT_ENTRY** combines the actions of **GET_NEXT_ENTRY** and **QUICK_LOOKUP** into one operation, and is considerably more efficient than those two procedures called serially. When traversing a subtree by calling **LOOKUP_NEXT_ENTRY**, the *level* field of the **Q_Info** record indicates the level of the object within the directory hierarchy. Objects in the root directory of a disk volume are at level zero.

2.10.20 MOUNT and UNMOUNT File System Calls

```
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 the volume's catalog to the File System. The name of the volume mounted is returned in the Vname parameter.

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

Processes

A *process* is an 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; they appear to be running simultaneously. Although processes can share code and data, each process has its own stack.

Only one process at a time can use the CPU. The *Scheduler* determines which process is active at a particular time. 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. The Scheduler then 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. A *running process* is actually executing instructions. A *ready process* is ready to execute but is being held back by the Scheduler. A *blocked process* is ignored by the Scheduler. It cannot continue its execution until something causes it to become 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.

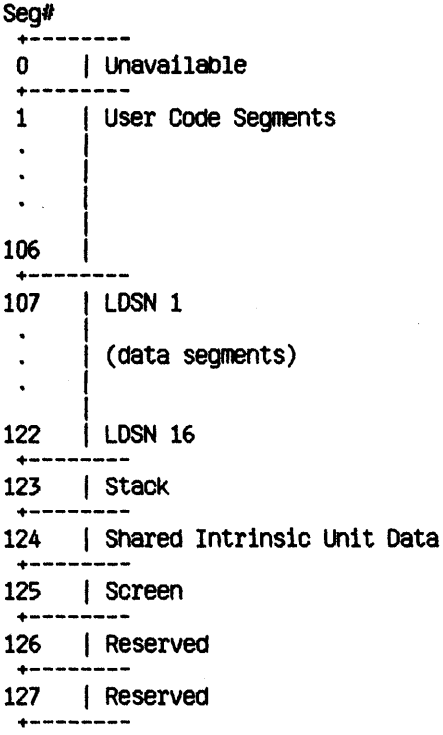


Figure 3-1
Process Address Space Layout

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

3.2 Process Hierarchy

When the system is first started, several system processes exist. At the base of the process hierarchy, shown in Figure 3-2, is the root process, which handles various internal Operating System functions. It has at least two sons: the Memory Manager process and the shell process.

The *Memory Manager process* handles code and data segment swapping.

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

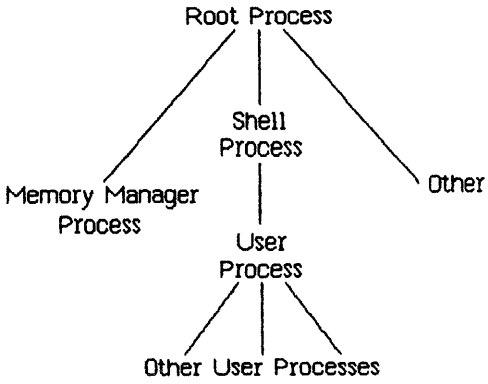


Figure 3-2
Process Tree

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 that 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 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 to control 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. 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 gets the CPU back when the Operating System is finished, except under the following conditions:

- The running process requests input or output. The Scheduler starts 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 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-priority 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 under one of the following conditions:

- It calls TERMINATE_PROCESS.
- It reaches an 'END.' statement.
- It is referred to in a KILL_PROCESS call.
- Its father process terminates.
- It runs into an abnormal condition.

When a process begins to terminate, a SYS_TERMINATE exception condition is signaled to the terminating process and all of the processes it has created. By means of the DECLARE_EXCEP_HDL call (described in Chapter 5), any process can create an exception handler to catch the terminate exception and clean up before terminating. The SYS_TERMINATE exception handler will be executed only once. If an error occurs while the handler is executing, the process terminates immediately.

A process can call KILL_PROCESS on any user process whose Proc_Id is known. TERMINATE_PROCESS, on the other hand, terminates the process that called it (and its descendants). TERMINATE_PROCESS also allows an event to be sent to the father of the terminating process if a local event channel was specified in the MAKE_PROCESS call.

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, below, creates a son process and lets it run for a while. It then gives the user 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 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;
```

3.8.1 MAKE_PROCESS Process System Call

```
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 son process is created when another process, the father process, calls MAKE_PROCESS. The son process executes the program identified by the pathname in **ProgFile**. If **ProgFile** is a null character string, the program name of the father process is used. A globally unique identifier for the son process is returned in **Proc_Id**.

Evnt_Chn_RefNum is a local event channel supplied by the father process. Event channels are discussed in Chapter 5. The Operating System uses the event channel identified by **Evnt_Chn_RefNum** to send the father process events regarding the son process (for example, SYS_SON_TERM). If **Evnt_Chn_RefNum** is zero, the father 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 for Pascal, this parameter is currently ignored.

Any error encountered during process creation is reported in **ErrNum**.

3.8.2 TERMINATE_PROCESS Process System Call

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

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

A process can be ended by **TERMINATE_PROCESS**. This call causes a SYS_TERMINATE exception to be signaled for the calling process and for 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 **ErrNum** 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 lockup 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 at the beginning 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 the process was created with a local event channel, an event is sent to the father if the process calls TERMINATE_PROCESS with a nil Event_Ptr or if the process terminates by a means other than calling TERMINATE_PROCESS. The event contains the termination cause in the first **longint** and zeroes in the remaining event text.

P_s_eventblk is a pointer to s_eventblk, 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.

3.8.3 INFO_PROCESS Process System Call

```
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 procedure on the next page 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;
```

3.8.4 KILL_PROCESS Process System Call

```
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.

3.8.5 SUSPEND_PROCESS Process System Call

```
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:

- Executing in user mode
- Stopped due to a **DELAY_TIME** call
- 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, by the timer (see **DELAY_TIME**), or 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.

3.8.6 ACTIVATE_PROCESS Process System Call

```
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 awaken only 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**.

3.8.7 SETPRIORITY_PROCESS Process System Call

```
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 priority value must be between 1 and 225. (Operating System processes execute with priorities between 226 and 255.) The higher the priority, the more likely the process is to be allowed to execute.

3.8.8 YIELD_CPU Process System Call

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

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

Background processes should use **YIELD_CPU** often to allow other processes to execute when they need to. Successive yields by processes of the same priority result in a "round robin" scheduling of the processes. If **To_Any** is true, **YIELD_CPU** causes the calling process to yield the CPU to any other ready process. 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 no process meets the **To_Any** criterion, the calling process simply continues execution.

3.8.9 MY_ID Process System Call

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

Memory Management

Every process has a set of code segments and data segments which are in physical memory when they are used. The logical address used by the process must be translated into the physical address used by the memory controller. This function 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 by the hardware and the Operating System.

A process can acquire additional data segments for uses such as heaps and interprocess 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.

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 that the program says it needs. It is the responsibility of the programmer to ensure 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 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 to unbind the data segment without closing 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 the same time, in addition to the stack. Each bound data segment is associated with a specific region of the address space by means of a Logical Data Segment Number (LDSN). See Figure 3-1 for an illustration of the address space of a process. 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 segment 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 a given 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 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 ensures that all logical data addresses referencing locations within the data segment are consistent for the processes sharing the segment. A shared data segment is permanent until explicitly killed by a process.

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 LDSNs available to map the segment. The process gives the desired segment size and the base LDSN to map the segment. The Memory Manager then uses ascending adjacent LDSNs to map successive 128 Kbyte chunks of the segment. The process must ensure that enough consecutive LDSNs 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 LDSNs. 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:

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

The Operating System ensures 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 System 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];
```

4.7.1 MAKE_DATASEG Memory Management System Call

```
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 are allocated to the segment. 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 ensure that enough consecutive LDSNs 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 ensure 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 created, it immediately becomes a member of the working set of the calling process. You can use **UNBIND_DATASEG** to free the LDSN.

4.7.2 KILL_DATASEG Memory Management System Call

```
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 it, the contents of the data segment are not written to disk and are lost when the segment is closed.

4.7.3 OPEN_DATASEG Memory Management System Call

```
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 LDSN given must not have a data segment currently bound to it. The segment's name is determined by the process that 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.

4.7.4 CLOSE_DATASEG Memory Management System Call

```
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;
```

4.7.5 FLUSH_DATASEG Memory Management System Call

```
FLUSH_DATASEG (Var ErrNum:Integer;
               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.

4.7.6 SIZE_DATASEG Memory Management System Call

```
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 old size plus delta size 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 ensure that all of its data segments are swappable before it relinquishes the attention of the processor.

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

4.7.7 INFO_DATASEG Memory Management System Call

```
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:Booleanm (* True if the calling process is the creator    *)
                   (* of the segment                                *);
 RWAccess:Boolean  (* True if the calling process has Write access *)
                   (* to segment                                    *)
END;
```

4.7.8 INFO_LDSN Memory Management System Call

```
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.

4.7.9 INFO_ADDRESS Memory Management System Call

```
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 that contains the address given is currently bound to the calling process, an error indication is returned in ErrNum.

4.7.10 MEM_INFO Memory Management System Call

```
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.

4.7.11 SETACCESS_DATASEG Memory Management System Call

```
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.

4.7.12 BIND_DATASEG and UNBIND_DATASEG Memory Management System Calls

```
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 LDSNs. **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 LDSNs must not be bound to a different data segment. **UNBIND_DATASEG** frees the associated LDSNs. A reference to the contents of an unbound segment gives unpredictable results. **OPEN_DATASEG** and **MAKE_DATASEG** define which LDSNs are associated with a given data segment.

Chapter 5
Exceptions and Events

Exceptions and Events

Processes have several ways to keep informed about the state of the system. Normal process-to-process communication and synchronization employ pipes, shared data segments, or events. Abnormal conditions, including those your program may define, employ exceptions (interrupts). Exceptions are signals to which the process can respond 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 a system-defined exception is *enabled*, the system looks for an associated user-defined handler. If none is found, the system invokes the default exception handler, which usually aborts the process that generated the exception. If a user-defined exception is enabled, the system invokes the associated user-defined exception handler. You create a new exception by declaring and enabling a handler for it.

If the state of the exception is *queued*, the exception is placed on a queue. When the exception is subsequently enabled, the 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 detects the occurrence of the exception, but the exception is neither honored nor queued. Note that ignoring a system-defined exception has 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 signaling 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). The default handler aborts the process.
- Value out of bounds (that is, range check error) or illegal string index (SYS_VALUE_OOB). The default handler aborts the process.
- Arithmetic overflow (SYS_OVERFLOW). The default handler aborts the process.
- Process termination (SYS_TERMINATE). This exception is signaled 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 **Data_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. **Data_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 dissociated 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 occurs only if you explicitly enable the exception.)

There is an exception-occurred flag, Ex_occurred_f, for every declared exception; it is set whenever the corresponding exception occurs. This flag can be examined and reset using the INFO_EXCEP system call. 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  (Environment_Ptr:p_env_blk;
                    Data_Ptr:p_ex_data);
VAR ErrNum:INTEGER;
BEGIN
(*Environment_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 for a SYS_TERMINATE exception, the stack is as shown in Figure 5-1.

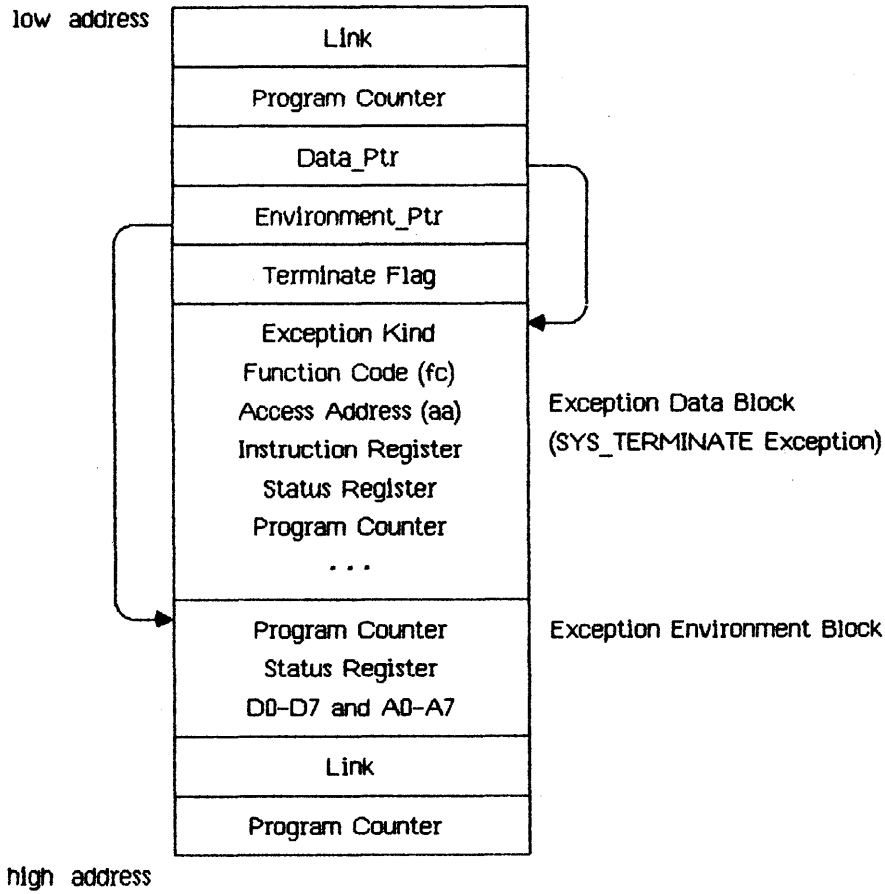


Figure 5-1
Stack at Exception Handler Invocation

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 signaled, 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 signaled. 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 be opened by user processes only 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_signaling, 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_signaling, 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,@Handler);
  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_namej_1,@Handler)
  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 it 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. For more information, 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);
```

5.7.1 DECLARE_EXCEP_HDL Exception Management System Call

```
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 is used. Any previously declared exception handler is dissociated by this call. The exception itself is automatically enabled.

If any **Excep_Name** exceptions are queued 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 dissociate your handler from the exception. If there is no system handler defined, the program that signals the exception receives an error 201.

5.7.2 DISABLE_EXCEP Exception Management System Call

```
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 receives the event, thus dequeuing the exception.

5.7.3 ENABLE_EXCEP Exception Management System Call

```
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.

5.7.4 INFO_EXCEP Exception Management System Call

```
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.

5.7.5 SIGNAL_EXCEP Exception Management System Call

```
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 signaled
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.

5.7.6 FLUSH_EXCEP Exception Management System Call

```
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;
```

5.8.1 MAKE_EVENT_CHN Event Management System Call

```
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.

5.8.2 KILL_EVENT_CHN Event Management System Call

```
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.

5.8.3 OPEN_EVENT_CHN Event Management System Call

```
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 signaled when an event arrives in the channel. **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.

5.8.4 CLOSE_EVENT_CHN Event Management System Call

```
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 cannot open it after it has been closed.

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

5.8.5 INFO_EVENT_CHN Event Management System Call

```
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;
```

5.8.6 WAIT_EVENT_CHN Event Management System Call

```
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 **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.

5.8.7 FLUSH_EVENT_CHN Event Management System Call

```
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 FLUSH_EVENT_CHN is called by a sender, it has no effect.

5.8.8 SEND_EVENT_CHN Event Management System Call

```
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 opened 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, which identify the sender, in the event text.

The **Interval** parameter indicates whether the event is a timed event.

NOTE

Timed events will not be supported in future releases of the Operating System. The **Interval** and **Clktime** parameters will be ignored in future releases. If you want your software to be upward-compatible, always set both fields of the **Interval** parameter to zero.

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;
```

5.9.1 DELAY_TIME Clock System Call

```
DELAY_TIME (Var  ErrNum:Integer;
                 Interval:Timestmp_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.**

5.9.2 GET_TIME Clock System Call

```
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 zero on return.

5.9.3 SET_LOCAL_TIME_DIFF Clock System Call

```
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.

5.9.4 CONVERT_TIME Clock System Call

```
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 in which direction the conversion is to go. If **To_Sys** is true, the system takes the time data in **Local_Time** and puts the corresponding system time in **Sys_Time.** If **To_Sys** is false, the system 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

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*. 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.

Several calls are provided that allow programs to request information about the configuration of the system.

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.

6.1.1 READ_PMEM Configuration System Call

READ_PMEM (Var ErrNum:Integer; Var PMrec:PMemRec)

ErrNum: Error code
PMrec: Contents of parameter memory

READ_PMEM returns the contents of parameter memory in **PMrec**. The contents of **PMrec** are not to be interpreted by the caller. This routine exists for the purpose of obtaining **PMrec** so that **PMrec** can be passed to the other configuration procedures described in this chapter.

6.1.2 GETNXTCONFIG Configuration System Call

```
GETNXTCONFIG  (Var ErrNum:Integer;
               Var NextEntry:Longint;
               Var PMrec:PMemRec;
               Var Config:ConfigDev)
```

```
ErrNum:    Error code
NextEntry: Enumeration index
PMrec:     Contents of parameter memory
Config:    Configuration entry
```

GETNXTCONFIG is used to enumerate device configuration information. **NextEntry** = 0 is passed by the caller to start the enumeration. After the first call to **GETNXTCONFIG**, the caller passes the previously returned value of **NextEntry** on each subsequent call to **GETNXTCONFIG**. The Operating System updates the value of **NextEntry** with each call. The enumeration is done using the caller's copy of parameter memory (obtained by calling **READ_PMEM**) which is input in **PMrec**. Upon return from the procedure, **Config** holds the next configuration record that was extracted from the copy of parameter memory. **ErrNum** = 799 is returned when no more configuration entries are available.

The **Config** record contains:

```
pos: cd_position;
nExtWords: byte; (*number of valid ExtWords following*)
ExtWords: array[1..3] of Integer;
DriverID: longint;
DevName: e_name;

where cd_position = record
                      slot, chan, dev: byte
                    end;
```

The **pos** record of three bytes indicates the position of the device being described. **DevName** is a character string representation of this position. The characteristics of the device can be obtained by calling **LOOKUP** and passing **-DevName** as input. Table 6-1 shows the device names, as well as the aliases, which may be substituted for **DevName** in any Operating System call.

Table 6-1
Device Names

Slot	Chan	Dev	DevName	Alias	Description
1	0	0	#1	SLOT1	Peripheral at slot 1
1	x	0	#1#x	SLOT1CHANx	at slot 1 channel x
1	x	y	#1#x#y	SLOT1CHANxDEVy	at slot 1 channel x device y
2	0	0	#2	SLOT2	Peripheral at slot 2
2	x	0	#2#x	SLOT2CHANx	at slot 2 channel x
2	x	y	#2#x#y	SLOT2CHANxDEVy	at slot 2 channel x device y
3	0	0	#3	SLOT3	Peripheral at slot 3
3	x	0	#3#x	SLOT3CHANx	at slot 3 channel x
3	x	y	#3#x#y	SLOT3CHANxDEVy	at slot 3 channel x device y
10	1	0	#10#1	RS232A	Serial Port A
10	2	0	#10#2	RS232B	Serial Port B
11	0	0	#11	PARAPORT	Parallel Port
12	0	0	#12	UPPER or PARAPORT	Hard disk on Lisa 2/10
13	0	0	#13	LOWER	Sony Drive
14	1	0	#14#1	UPPER	Upper Floppy on Lisa 1
14	2	0	#14#2	LOWER	Lower Floppy on Lisa 1
15	1	0	#15#1	ALTCONSOLE	Alternate Console
15	2	0	#15#2	MAINCONSOLE	Main Console

ExtWords contains optional extension words. If the device is a printer, **ExtWords[1]** contains the following:

```
RECORD
 printer_flag: boolean; (* = true(1) *)
 default_flag: boolean; (* true if it's the default printer*)
 printerID: 14 bits     (* unique printer ID:
                             32 = Imagewriter / || DMP
                             33 = Daisy Wheel Printer
                             35 = Ink Jet Printer *)
END;
```

DriverID contains the unique driver ID:

```
32 = Serial Cable
33 = Parallel Cable
34 = 2 Port Card
35 = ProFile
36 = Sony
37 = Priam Card
38 = Priam Disk
39 = Archive Tape
40 = Console
42 = Modem A
```

6.1.3 MACH_INFO Configuration System Call

```
MACH_INFO   (Var ErrNum:Integer;
             Var The_info:Minfo)
```

ErrNum: Error code
The_info: Type of Lisa being used

MACH_INFO returns an array, **The_info**, showing the CPU board, I/O board and memory board in use:

```
minfo = RECORD
          cpu_board, io_board, mem_size: longint
        END;
```

cpu_board always returns 0. **mem_size** returns the number of bytes in memory. **io_board** returns:

0 = Lisa 1
1 = Lisa 2/10
2 = Lisa 2, Lisa 2/5, or Lisa 1 upgraded to use micro diskettes.

6.1.4 CARDS_EQUIPPED Configuration System Call

```
CARDS_EQUIPPED   (Var ErrNum:Integer;
                      Var In_Slot:Slot_array)
```

ErrNum: Error code
In_Slot: Identifies the types of cards configured

CARDS_EQUIPPED 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 integer;
```

where the array values may contain:

0 = no card present
2 = 2-port parallel card
5 = Priam card

6.1.5 OSBOOTVOL Configuration System Call

OSBOOTVOL (Var ErrNum:Integer; var VolName: e_name);

ErrNum: Error code
VolName: Identifies the device name for the boot volume

OSBOOTVOL returns the device name of the boot volume. This port might not be the port configured for the boot volume, since it is possible for the user to override the default boot volume. Characteristics about the device can be obtained by calling **LOOKUP** and passing **VolName**.

Appendixes

Appendix A
Operating System Interface Unit

```
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 pmByteUnique;


(* 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. Signaled when the TRAPV instruction is executed and the overflow condition is on.
SYS_VALUE_OOB	Value-out-of-bound exception. Signaled when the CHK instruction is executed and the value is less than 0 or greater than upper bound.
SYS_ZERO_DIV	Division by zero exception. Signaled when the DIVS or DIVU instruction is executed and the divisor is zero.
SYS_TERMINATE	Termination exception. Signaled 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
Error Messages

-6081 End of exec file input
-6004 Attempt to reset text file with typed-file type
-6003 Attempt to reset nontext file with text type
-1885 ProFile not present during driver initialization
-1882 ProFile not present during driver initialization
-1840 Packet ended in a resumable state (Archive).
-1293 Object is not password protected.
-1176 Data in the object have 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 are 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 and altered
-149 Process was created, but the specified program file has been scavenged and altered
-125 Specified 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 Cannot 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 the 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 Cannot 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_Excep_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 Insufficient memory for operation
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 Cannot perform explicit stack expansion due to lack of memory
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 nonconfigured device driver
606 Cannot find sector on floppy diskette (disk unformatted)
608 Illegal length or disk address for transfer
609 Call to nonconfigured 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 Cannot 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 are 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
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 do not 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 diskette loader on a ProFile
735 SYSTEM.LLD is missing
736 SYSTEM.UNPACK is missing
737 Unpack of SYSTEM.OS with SYSTEM.UNPACK failed
750 Position specified is out of range.
751 No device exists at the requested position.
752 Can't perform requested function while device is busy.
753 Specified position is not a terminal node.
754 Built-in devices cannot be configured.
755 Isolated positions cannot be configured.
756 The specified position is already configured.
757 Parallel Port doesn't exist on this type of machine.
758 No room in memory for more devices.
790 Can't get buffer space to load configurable driver.
791 Configurable driver code file is not executable.
792 Can't get memory space for a configurable driver.
793 I/O error reading configurable driver file.
794 Configurable driver code file not found.
795 Configurable driver has more than one segment.
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
840 The allocation map of this file is truncated.
841 Unallocated s-file or I/O error
843 Requested exact fit, but one could not 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
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
868 Bad unit number
869 Unit already mounted (mount)/no unit mounted
870 No sysglobal space for DCB or MDDF
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
922 Invalid label size
926 Pathname invalid or no such device
927 Invalid label size
941 Pathname invalid or no such device
944 Object is not a file
945 File is not in the killed state
946 Pathname invalid or no such device
947 Not enough space in syslocal for File System refdb
948 Entry not found in specified catalog
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
952 Bad refnum
954 Bad refnum
955 Read access not allowed to specified object
956 Attempt to position FMARK past LEOF not allowed
957 Negative request count is illegal
958 Nonsequential access is not allowed
959 System resources exhausted
960 Error writing to pipe while an unsatisfied read was pending
961 Bad refnum
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 Nonsequential access is not allowed
968 Bad refnum
971 Pathname invalid or no such device
972 Entry not found in specified catalog
974 Bad refnum
977 Bad refnum
978 Page count is nonpositive
979 Not a block-structured device

981 Bad refnum
982 No space has been allocated for specified file
983 Not a block-structured device
985 Bad refnum
986 No space has been allocated for specified file
987 Not a block-structured device
988 Bad refnum
989 Caller is not a reader of the pipe
990 Not a block-structured device
994 Invalid refnum
995 Not a block-structured device
999 Asynchronous read was unblocked before it was satisfied
1002 Invalid Device_Control call for device (Priam).
1003 Unable to get SysGlobal space for disk operation (Priam).
1021 Pathname invalid or no such entry
1022 No such entry found
1023 Invalid newname, check for '-' in string
1024 New name already exists in catalog
1031 Pathname invalid or no such entry
1032 Invalid transfer count
1033 No such entry found
1041 Pathname invalid or no such entry
1042 Invalid transfer count
1043 No such entry found
1051 No device or volume by that name
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
1062 No volume is mounted on device
1071 Not a valid or mounted volume for working directory
1091 Pathname invalid or no such entry
1092 No such entry found
1101 Invalid device name
1121 Invalid device, not mounted, or catalog is damaged
1122 No space for catalog scan buffer (Reset_Catalog).
1124 No space for catalog scan buffer (Get_Next_Entry).
1128 Invalid pathname, device, or volume not mounted
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 diskette device
1165 No volume is mounted on the device
1166 A valid volume is already mounted on the device
1167 Not a block-structured device
1168 Device name is invalid
1169 Could not access device before initialization using default device parameters
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 its 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 its 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
1193 Premature end of file encountered
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 Cannot write boot tracks on that device
1216 Could not create/close internal buffer
1217 Boot track program has too many code segments
1218 Could not find configuration information entry
1219 Could not 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
1281 Pathname is invalid because device or object is not present.
1282 Pathname syntax is invalid.
1283 Interior pathname component does not specify a directory object.
1284 Directory cannot be deleted because it is not empty.
1285 Operation is not allowed on a volume with a flat catalog.
1286 Operation is not allowed on a directory object.
1287 Cannot allocate SysLocal space for the directory scan stack.
1288 Directory tree is inconsistent.
1289 Operation not allowed against a volume or device (Quick_Lookup)
1290 The directory that contained the file has been deleted (Unkill_File)
1294 Supplied password does not match the password on the object.
1295 The allocation map of this file is damaged and cannot be read.
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
1840 Unable to initialize disk drive (Priam).
1841 Error writing to disk (Priam) / Error reading from tape (Archive).
1842 Error reading from disk (Priam) / Error writing to tape (Archive).

1843 Error controlling tape (Archive).
1844 Packet ended in a non-resumable state (Archive).
1845 Packet command had an error (Archive).
1882 Bad response from ProFile
1885 ProFile timeout error
1998 Invalid parameter address
1999 Bad refnum
6001 Attempt to access unopened file
6002 Attempt to reopen a file which is not closed using an open FIB (file info block)
6003 Operation incompatible with access mode with which file was opened
6004 Printer offline
6005 File record type incompatible with character device (must be byte sized)
6006 Bad integer (read)
6010 Operation incompatible with file type or access mode
6081 Premature end of exec file
6082 Invalid exec (temporary) file name
6083 Attempt to set prefix with null name
6090 Attempt to move console with exec or output file open
6101 Bad real (read)
6151 Attempt to reinitalize heap already in use
6152 Bad argument to NEW (negative size)
6153 Insufficient memory for NEW request
6154 Attempt to RELEASE outside of heap

Operating System Error Codes

The error codes listed below are generated only when a nonrecoverable error occurs while in Operating System code.

10050 Request block is not chained to a PCB (Unblk_Req)
10051 Bld_Req is called with interrupts off
10100 An error was returned from SetUp_Directory or a Data Segment routine (Setup_IUInfo)
10102 Error > 0 trying to create shell (Root)
10103 Sem_Count > 1 (Init_Sem)
10104 Could not open event channel for shell (Root)
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 reason other than I/O while scheduling is disabled (SimpleScheduler)
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
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 range error
10212 Error in Term_Def_Hdl from Enable_Excep
10213 Error in Force_Term_Excep, no space in Enq_Ex_Data

10401 Error from Close_Event_Chn in Ec_Cleanup
10582 Unable to get space in Freeze_Seg
10590 Fatal memory 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 allocated segment (Alloc_Mem)
10598 Attempting to allocate a nonfree memory region (Take_Free)
10599 Disk I/O error while swapping in an OS code segment.
10600 Error attempting to make timer pipe
10601 Error from Kill_Object of an 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
10605 Error during allocate space for timer pipe, or interrupt from nonconfigured device
10609 Interrupt from nonconfigured device
10610 Error from info about timer pipe
10611 Spurious interrupt from floppy drive #2
10612 Spurious interrupt from floppy drive #1, or 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
10619 No sysglobal space for timer request block
10624 Attempt to shut down floppy disk controller while drive is still busy
10637 Not enough memory to initialize system timeout drives
10675 Spurious timeout on console driver
10699 Spurious timeout on parallel printer driver
10700 Mismatch between loader version number and Operating System version number
10701 OS exhausted its internal space during startup
10702 Cannot make system process
10703 Cannot kill pseudo-outer process
10704 Cannot create driver
10706 Cannot initialize floppy disk driver
10707 Cannot initialize the File System volume
10708 Hard disk mount table unreadable
10709 Cannot map screen data
10710 Too many slot-based devices
10724 The boot tracks do not know the right File System version
10725 Either damaged File System or damaged contents
10726 Boot device read failed

10727 The OS will not fit into the available memory
10728 SYSTEM.OS is missing
10729 SYSTEM.CONFIG is corrupt
10730 SYSTEM.OS is corrupt
10731 SYSTEM.DEBUG or SYSTEM.DEBUG2 is corrupt
10732 SYSTEM.LLD is corrupt
10733 Loader range error
10734 Wrong driver is found. For instance, storing a diskette loader on a ProFile
10735 SYSTEM.LLD is missing
10736 SYSTEM.UNPACK is missing
10737 Unpack of SYSTEM.OS with SYSTEM.UNPACK failed
10738 Can't find a required driver for the boot device.
10739 Can't load a required driver for the boot device.
10740 Boot device won't initialize.
10741 Can't boot from a serial device.
11176 Found a pending write request for a pipe while in Close_Object when it is called by the last writer of the pipe
11177 Found a pending read request for a pipe while in Close_Object when it is called by the (only possible) reader of the pipe
11178 Found a pending read request for a pipe while in Read_Data from the pipe
11180 Found a pending write request for a pipe while in Write_Data to the pipe
118xx Error xx from diskette ROM (See OS errors 18xx)
11901 Call to Getspace or Relspace with a bad parameter, or free pool is bad

Appendix E
FS_INFO Fields

* *defined for mounted or unmounted devices*
$ *defined for mounted devices only*
All other fields are defined for mounted block-structured devices only.

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.

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 to 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.

Index

Please note that the topic references in this Index are *by section number.*

Apple publications would like to learn about readers and what you think about this manual in order to make better manuals in the future. Please fill out this form, or write all over it, and send it to us. We promise to read it.

How are you using this manual?
[ ] learning to use the product [ ] reference [ ] both reference and learning

[ ] other_____

Is it quick and easy to find the information you need in this manual?
[ ] always [ ] often [ ] sometimes [ ] seldom [ ] never

Comments_____

What makes this manual easy to use?_____

_____

What makes this manual hard to use?_____

_____

What do you like most about the manual?_____

_____

What do you like least about the manual?_____

_____

Please comment on, for example, accuracy, level of detail, number and usefulness of examples, length or brevity of explanation, style, use of graphics, usefulness of the index, organization, suitability to your particular needs, readability.

_____

_____

_____

What languages do you use on your Lisa? (check each)

[ ] Pascal [ ] BASIC [ ] COBOL [ ] other_____

How long have you been programming?

[ ] 0-1 years [ ] 1-3 [ ] 4-7 [ ] over 7 [ ] not a programmer

What is your job title?_____

Have you completed:

[ ] high school [ ] some college [ ] BA/BS [ ] MA/MS [ ] more

What magazines do you read?_____

_____

_____

Other comments (please attach more sheets if necessary)_____

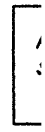
_____

_____

_____

FOLD

FOLD



apple computer

POS Publications Department
20525 Mariani Avenue
Cupertino, California 95014

TAPE OR STAPLE

The OEMSysCall Unit

Contents

The OEMSysCall Unit

1 Introduction

The OEMSysCall unit provides interfaces to privileged procedures within the Lisa Operating System. These privileged procedures offer facilities that fall into two categories: disk volume management and file password protection.

Disk Volume Management

The OEMSysCall unit includes procedures to

- Initialize a disk volume.
- Eject a removable disk volume.
- Scavenge a disk volume.
- Determine if two disk volumes are identical.

File Password Protection

A file may be protected from unauthorized access by associating a password with it. Password protection prevents a file from being opened, killed, or renamed without presentation of the proper password. Other operations (e.g., Lookup, Read_Label, etc.) are unaffected by the presence of a password protecting the specified file. The OEMSysCall unit includes procedures to

- Open a password-protected file.
- Delete a password-protected file.
- Rename a password-protected file.
- Change the password associated with a file.
- Verify the password associated with a file.

2 OEMSysCall Routines

2.1 Init_Vol

```
Init_Vol (var ecode : integer;
              devName : e_name;
              volName : e_name;
              password : e_name)
```

```
ecode:    Error indication (common errors are listed below)
devName:  Name of the device to initialize
volName:  Name to assign to the new disk volume
password: Password to assign to the new disk volume
```

Initialize the volume on the specified device. The volume is assigned the name and password **volName** and **password**. Volume passwords are currently not supported by the Lisa file system. The volume may not be mounted on the device at the time of the call.

Common errors:

618	Cannot format the volume (make sure a diskette is in the drive).
971	Device name is invalid (check configuration).
1167	Device is not a disk.
1169	Could not default mount the volume in order to perform initialization.
1171	Volume name contains the dash, "-", character.
1172	No space in system heap for the volume allocation map of the new volume.
1390	Volume is mounted on the device.

2.2 EjectVol

```
EjectVol (var ecode : integer;
          devName : e_name)
```

ecode: Error indication (common errors are listed below)
devName: Name of the device from which to eject media

Eject the removable disk media from the specified device. The device must support ejectable media, and the volume may not be mounted on the device at the time of the call.

Common errors:

614	No diskette present in the drive.
971	Device name is invalid (check configuration).
1164	Device does not support ejectable media.
1390	Volume is mounted on the device.

2.3 ScavengeVol

```
ScavengeVol (var ecode : integer;
             devName : e_name)
```

ecode: Error indication (common errors are listed below)
devName: Name of the device to scavenge

Scavenge the volume on the specified device. The volume may not be mounted on the device at the time of the call.

Common errors:

614	No diskette present in the drive.
971	Device name is invalid (check configuration).
1225	Scavenger aborted.
1227	Device is not a disk.
1231	Scavenger heap overflow.
1237	Unable to repair the volume directory structure.
1240	Volume is not in a Lisa file system format.
1390	Volume is mounted on the device.

2.4 VerifyVol

```
VerifyVol (var ecode : integer;
               sourceDev : e_name;
               destinDev : e_name;
               bufAddr : longint;
               bufSize : longint)
```

```
ecode:     Error indication (common errors are listed below)
sourceDev: Name of the device being verified
destinDev: Name of the device to verify against
bufAddr:   Address of the buffer
bufSize:   Size of the buffer in bytes
```

Compare the volume on **sourceDev** with the volume on **destinDev**. The volumes are compared track by track. The memory buffer used during the comparison is supplied by the caller and is described by its starting address **bufAddr** and length **bufSize**. The buffer must be at least large enough to accommodate two disk blocks of 536 bytes each (i.e., 1072 bytes). Neither the source volume nor the destination volume may be mounted at the time of the call. The error indication **ecode** is zero if the volumes are identical, and 1393 if they differ.

Common errors:

```
 614   No diskette present in the drive.
 971   Source or destination device name is invalid (check
       configuration).
1167   Source or destination device is not a disk.
1390   Volume is mounted on the source or destination
       device.
1392   Supplied buffer is too small (bufSize < 1072).
1393   Volumes are not identical.
```

2.5 MakeSecure

```
MakeSecure (var ecode : integer;
            var path : pathname;
            var password : e_name)

ecode:    Error indication (common errors are listed below)
path:     Name of the new file
password: Password to be associated with the new file
```

Create a new file protected by the specified password. This procedure behaves the same as Make_File.

Common errors:

```
854    Volume s-file list is full.
855    Cannot allocate disk space for the file leader.
890    File already exists.
891    Volume catalog is full.
892    File name is illegal (a file name may not contain
       the dash, "-", character).
921    Pathname is invalid.
```

2.6 KillSecure

```
KillSecure (var ecode : integer;
            var path : pathname;
            var password : e_name)

ecode:    Error indication (common errors are listed below)
path:     Name of the object to be deleted
password: Password associated with the object
```

Delete the file with the specified name and password. The deletion is not allowed if **password** does not match the password assigned to the file. This procedure behaves the same as Kill_Object.

Common errors:

-1293	Warning: the file was not password protected. The kill operation completes normally.
894	File cannot be found.
895	File name is illegal.
896	File safety switch is set (the file is protected against deletion).
1294	Supplied password does not match the password protecting this file.
1298	File cannot be accessed because its s-list entry is damaged.

2.7 OpenSecure

```
OpenSecure (var ecode : integer;
            var path : pathname;
            var refnum : integer;
                manip : mset;
            var password : e_name)
```

```
ecode:    Error indication (common errors are listed below)
path:     Name of object to be opened
refnum:   Reference number for the object
manip:    Set of access types
password: Password associated with the object
```

Open the file with the specified name and password. The open is not done if **password** does not match the password assigned to the file. This procedure behaves the same as Open.

Common errors:

-1173	Warning: this file was last closed by the Operating System.
-1174	Warning: this file was open during a system crash.
-1175	Warning: this file has been reconstructed by the scavenger.
-1176	Warning: the *contents* of this file has been reconstructed by the scavenger.
-1293	Warning: the file was not password protected. The open operation completes normally.
871	File cannot be accessed because its s-list entry is damaged.
872	No space in system heap for File Control Block.
873	File is open for *private* access by another process.
946	Pathname is invalid.
947	No space in the system heap for File Refnum Descriptor Block.
948	File cannot be found.
949	Request for *private* access is disallowed because the file is open for access by another process.
1130	Open request violates software theft protection policy.
1294	Supplied password does not match the password protecting this file.

2.8 RenameSecure

```
RenameSecure (var ecode : integer;
              var path : pathname;
              var newName : e_name;
              var password : e_name)
```

```
ecode:    Error indication (common errors are listed below)
path:     Name of the object to be renamed
newName:  New name for the object
password: Password associated with the object
```

Rename the file with the specified name and password. The rename is not done if **password** does not match the password assigned to the file. This procedure behaves the same as Rename_Entry.

Common errors:

-1293	Warning: the file was not password protected. The rename operation completes normally.
1021	Pathname is invalid.
1022	File cannot be found.
1023	New file name is illegal (a file name cannot contain the dash, "-", character).
1024	File having the new name already exists.
1294	Supplied password does not match the password protecting this file.
1296	File name string variable has bad length byte.
1297	File cannot be accessed because its leader is damaged.
1298	File cannot be accessed because its s-list entry is damaged.

2.9 VerifyPassword

```
VerifyPassword (var ecode : integer;
                var path : pathname;
                var password : e_name)
```

ecode: Error indication (common errors are listed below)
path: Name of the file whose password is to be verified
password: Password to be verified

Compare the specified password with the password protecting the specified file. The error indication **ecode** is zero if the passwords are identical, and 1294 if they differ.

Common errors:

-1293	Warning: the file was not password protected. The verify operation completes normally.
1091	Pathname is invalid.
1092	File cannot be found.
1297	File cannot be accessed because its leader is damaged.
1298	File cannot be accessed because its s-list entry is damaged.
1294	Supplied password does not match the password protecting this file.

2.10 ChangePassword

```
ChangePassword (var ecode : integer;
                var path : pathname;
                var oldPassword : e_name;
                var newPassword : e_name)
```

```
ecode:       Error indication (common errors are listed
             below)
path:        Name of the file whose password is to be
             changed
oldPassword: Current password associated with the file
newPassword: New password to be associated with the file
```

Change the password associated with the specified file. The change is not done if **oldPassword** does not match the password assigned to the file. This call may be used to assign a password to a file for the first time.

Common errors:

```
-1293   Warning: the file was not password protected.  The
        change operation completes normally.
 1091   Pathname is invalid.
 1092   File cannot be found.
 1297   File cannot be accessed because its leader is
        damaged.
 1298   File cannot be accessed because its s-list entry is
        damaged.
 1294   Supplied old password does not match the password
        protecting this file.
```

3 Interface

```
UNIT OEMsyscall;

INTERFACE

 USES
   (*$U Syscall.obj *) syscall,
   (*$U Psyscall.obj *) psyscall;

 procedure EJECTVOL(var errnum:integer; devname:e_name);

 procedure SCAVENGEVOL(var errnum:integer; devname:e_name);

 procedure INIT_VOL(var errnum:integer; devname:e_name; volname:e_name;
                    password:e_name);

 procedure VERIFYVOL(var errnum:integer; sourcedev:e_name; destdev:e_name;
                     buffaddr:longint; buffsize:longint);

 procedure MAKESECURE(var errnum:integer; var path:pathname; var
                      passwd:e_name);

 procedure KILLSECURE(var errnum:integer; var path:pathname; var
                      passwd:e_name);

 procedure OPENSECURE(var errnum:integer; var path:pathname; var refnum:integer;
                      manip:mset; var passwd:e_name);

 procedure RENAMESECURE(var errnum:integer; var path:pathname; var
                        newname:e_name; var passwd:e_name);

 procedure VERIFYPASSWORD(var errnum:integer; var path:pathname; var
                          passwd:e_name);

 procedure CHANGEPASSWORD(var errnum:integer; var path:pathname; var
                          oldpasswd:e_name; var newpasswd:e_name);
```

The Standard Apple Numeric Environment

Contents

The Standard Apple Numeric Environment

1 Introduction

This manual describes the Standard Apple Numeric Environment (SANE). Apple supports SANE on several current products and plans to support SANE on future products. SANE gives you access to numeric facilities unavailable on almost any computer of the early 1980s--from microcomputers to extremely fast, extremely expensive supercomputers. The core features of SANE are not exclusive to Apple; rather they are taken from Draft 10.0 of Standard 754 for Binary Floating-Point Arithmetic [10] as proposed to the Institute of Electrical and Electronics Engineers (IEEE). Thus SANE is one of the first widely available products with the arithmetic capabilities destined to be found on the computers of the mid-1980s and beyond.

The IEEE Standard specifies standardized data types, arithmetic, and conversions, along with tools for handling limitations and exceptions, that are sufficient for numeric applications. SANE supports all requirements of the IEEE Standard. SANE goes beyond the specifications of the Standard by including a data type designed for accounting applications and by including several high-quality library functions for financial and scientific calculations.

IEEE arithmetic was specifically designed to provide advanced features for numerical analysts without imposing extra burden on casual users. (This is an admirable but rarely attainable goal: text editors and word processors, for example, typically suffer increased complexity with added features, meaning more hurdles for the novice to clear before completing even the simplest tasks.) The independence of elementary and advanced features of the IEEE arithmetic was carried over to SANE.

2 Data Types

SANE provides three *application* data types (single, double, and comp) and the *arithmetic* type (extended). Single, double, and extended store floating-point values and comp stores integral values.

The *extended* type is called the arithmetic type because, to make expression evaluation simpler and more accurate, SANE performs all arithmetic operations in extended precision and delivers arithmetic results to the extended type. *Single, double,* and *comp* can be thought of as space-saving storage types for the extended-precision arithmetic. (In this manual, we shall use the term *extended precision* to denote both the extended precision and the extended range of the extended type.)

All values representable in single, double, and comp (as well as 16-bit and 32-bit integers) can be represented exactly in extended. Thus values can be moved from any of these types to the extended type and back without any loss of information.

2.1 Choosing a Data Type

Typically, picking a data type requires that you determine the trade-offs between

- Fixed- or floating-point form,
- Precision,
- Range,
- Memory usage, and
- Speed.

The precision, range, and memory usage for each SANE data type are shown in Table 2-1. Effects of the data types on performance (speed) vary among the implementations of SANE. (See Section 4 for information on conversion problems relating to precision.)

Most accounting applications require a counting type that counts things (pennies, dollars, widgets) exactly. Accounting applications can be implemented by representing money values as integral numbers of cents or mils, which can be stored exactly in the storage format of the *comp* (for computational) type. The sum, difference, or product of any two comp values is exact if the magnitude of the result does not exceed $2^{63} - 1$ (that is, 9,223,372,036,854,775,807). This number is larger than the U.S. national debt expressed in Argentine pesos. In addition, comp values (such as the results of accounting computations) can be mixed with extended values in floating-point computations (such as compound interest).

Arithmetic with comp-type variables, like all SANE arithmetic, is done internally using extended-precision arithmetic. There is no loss of precision, as conversion from comp to extended is always exact. Space can be saved

by storing numbers in the comp type, which is 20 percent shorter than extended. Nonaccounting applications will normally be better served by the floating-point data formats.

2.2 Values Represented

The floating-point storage formats (single, double, and extended) provide binary encodings of a *sign* (+ or –), an *exponent*, and a *significand*. A represented number has the value

$$\pm\text{significand} * 2^{\text{exponent}}$$

where the significand has a single bit to the left of the binary point (that is, $0 \leq \text{significand} < 2$).

2.3 Range and Precision of SANE Types

This table describes the range and precision of the numeric data types supported by SANE. Decimal ranges are expressed as chopped two-digit decimal representations of the exact binary values.

Table 2-1
SANE Types

Type class	Application			Arithmetic
Type identifier	Single	Double	Comp	Extended
Size (bytes:bits)	4:32	8:64	8:64	10:80
Binary exponent range				
Minimum	-126	-1022	----	-16383
Significand precision				
Bits	24	53	63	64
Decimal digits	7-8	15-16	18-19	19-20
Decimal range (approximate)				
Min negative	-3.4E+38	-1.7E+308	≈-9.2E18	-1.1E+4932
Max neg norm	-1.2E-38	-2.3E-308		-1.7E-4932
Max neg denorm*	-1.5E-45	-5.0E-324		-1.9E-4951
Min pos denorm*	1.5E-45	5.0E-324		1.9E-4951
Min pos norm	1.2E-38	2.3E-308		1.7E-4932
Max positive	3.4E+38	1.7E+308	≈ 9.2E18	1.1E+4932
Infinities*	Yes	Yes	No	Yes
NaNs*	Yes	Yes	Yes	Yes

* *Denorms* (*denormalized numbers*), *NaNs* (*Not-a-Number*), and *infinities* are defined in Section 7.

Usually numbers are stored in a *normalized* form, to afford maximum precision for a given significand width. Maximum precision is achieved if the high order bit in the significand is 1 (that is, 1 ≤ significand < 2).

Example

In Single, the largest representable number has

significand	=	$2 - 2^{-23}$
	=	$1.11111111111111111111111_2$
exponent	=	127
value	=	$(2 - 2^{-23}) * 2^{127}$
	≈	$3.403 * 10^{38}$

the smallest representable positive normalized number has

significand	=	1
	=	$1.00000000000000000000000_2$
exponent	=	-126
value	=	$1 * 2^{-126}$
	≈	$1.175 * 10^{-38}$

and the smallest representable positive denormalized number (see Section 7) has

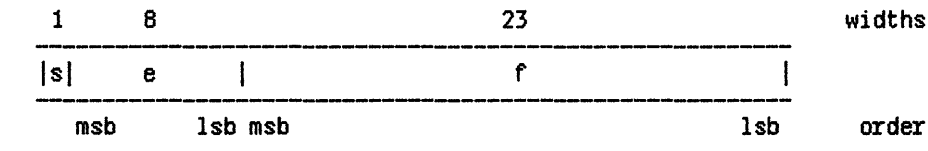
significand	=	2^{-23}
	=	$0.00000000000000000000001_2$
exponent	=	-126
value	=	$2^{-23} * 2^{-126}$
	≈	$1.401 * 10^{-45}$

2.4 Formats

This section shows the formats of the four SANE numeric data types. These are pictorial representations and may not reflect the actual byte order in any particular implementation.

Single

A 32-bit single format number is divided into three fields as shown below.



The value v of the number is determined by these fields as follows:

if 0 < e < 255, then $v = (-1)^s * 2^{(e-127)} * (1.f)$;

if e = 0 and f ≠ 0, then $v = (-1)^s * 2^{(-126)} * (0.f)$;

if e = 0 and f = 0, then $v = (-1)^s * 0$;

if e = 255 and f = 0, then $v = (-1)^s * \infty$;

if e = 255 and f ≠ 0, then v is a NaN.

See Section 7 for information on the contents of the f field for NaNs.

Double
A 64-bit double format number is divided into three fields as shown below.

The value v of the number is determined by these fields as follows:

if 0 < e < 2047, then $v = (-1)^s * 2^{(e-1023)} * (1.f)$;

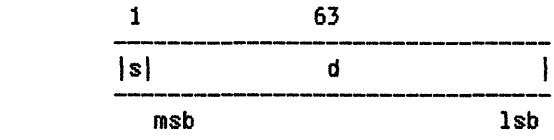
if e = 0 and f ≠ 0, then $v = (-1)^s * 2^{(-1022)} * (0.f)$;

if e = 0 and f = 0, then $v = (-1)^s * 0$;

if e = 2047 and f = 0, then $v = (-1)^s * \infty$;

if e = 2047 and f ≠ 0, then v is a NaN.

Comp
A 64-bit comp format number is divided into two fields as shown below.

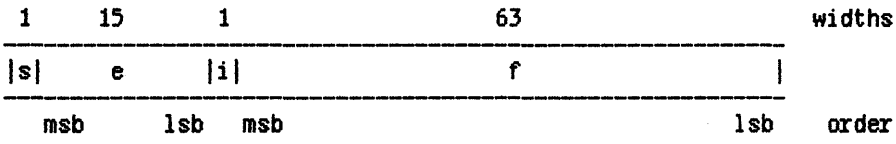


The value v of the number is determined by these fields as follows:

if s = 1 and d = 0, then v is the unique comp NaN;

otherwise, v is the two's-complement value of the 64-bit representation.

Extended
An 80-bit extended format number is divided into four fields as shown below.



The value v of the number is determined by these fields as follows:

if 0 <= e < 32767, then $v = (-1)^s * 2^{(e-16383)} * (i.f)$;

if e = 32767 and f = 0, then $v = (-1)^s * \infty$, regardless of i;

if e = 32767 and f ≠ 0, then v is a NaN, regardless of i.

3 Arithmetic Operations

SANE provides the basic arithmetic operations for the SANE data types:

- Add.
- Subtract.
- Multiply.
- Divide.
- Square root.
- Remainder.
- Round to integral value.

All the basic arithmetic operations produce the best possible result: The mathematically exact result coerced to the precision and range of the extended type. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE Standard (see Section 8). See Sections 9 and 10 for auxiliary operations and higher-level functions supported by SANE.

3.1 Remainder

Generally, remainder (and mod) functions are defined by the expression

```
x rem y = x - y * n
```

where n is some integral approximation to the quotient x/y. This expression can be found even in the conventional integer-division algorithm:

```
                 n          (integral quotient approximation)
(divisor)    y) x           (dividend)
                    y * n
                 ---------
                 x - y * n  (remainder)
```

SANE supports the remainder function specified in the IEEE Standard:

When y ≠ 0, the remainder r = x rem y is defined regardless of the rounding direction by the mathematical relation r = x - y * n, where n is the integral value nearest the exact value x/y; whenever |n - x/y| = 1/2, n is even. The remainder is always exact. If r = 0, its sign is that of x.

Example 1

Find 5 rem 3 . Here x = 5 and y = 3. Since 1 < 5/3 < 2 and since 5/3 = 1.66666... is closer to 2 than to 1, n is taken to be 2, so

```
5 rem 3  =  r  =  5 - 3 * 2  =  -1
```

Example 2

Find 7.0 rem 0.4 . Since 17 < 7.0/0.4 < 18 and since 7.0/0.4 = 17.5 is equally close to both 17 and 18, n is taken to be the even quotient, 18. Hence,

```
7.0 rem 0.4  =  r  =  7.0 - 0.4 * 18  =  -0.2
```

The IEEE remainder function differs from other commonly used remainder and mod functions. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. All the other remainder functions can be constructed from the IEEE remainder.

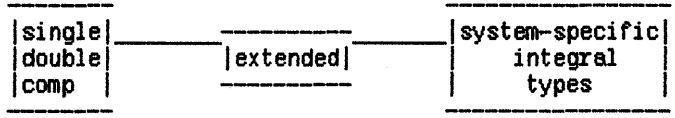
3.2 Round to Integral Value

An input argument is rounded according to the current rounding direction to an integral value and delivered to the extended format. For example, 12345678.875 rounds to 12345678.0 or 12345679.0. (The rounding direction, which can be set by the user, is explained fully in Section 8.)

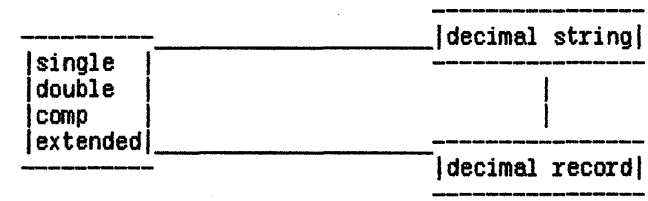
Note that, in each floating-point format, all values of sufficiently great magnitude are integral. For example, in single, numbers whose magnitudes are at least 223 are integral.

4 Conversions

SANE provides conversions between the extended type and each of the other SANE types (single, double, and comp). A particular SANE implementation will provide conversions between extended and those numeric types supported in its particular larger environment. For example, a Pascal implementation will have conversions between extended and the Pascal integer type.



SANE implementations also provide either conversions between decimal strings and SANE types, or conversions between a decimal record type and SANE types, or both. Conversions between decimal records and decimal strings may be included too.



4.1 Conversions between Extended and Single or Double

A conversion to extended is always exact. A conversion from extended to single or double moves a value to a storage type with less range and precision, and sets the overflow, underflow, and inexact exception flags as appropriate. (See Section 8 for a discussion of exception flags.)

4.2 Conversions to Comp and Other Integral Formats

Conversions to integral formats are done by first rounding to an integral value (honoring the current rounding direction) and then, if possible, delivering this value to the destination format. If the source operand of a conversion from extended to comp is a NaN, an infinity, or out-of-range for the comp format, then the result is the comp NaN and for infinities and values out-of-range, the invalid exception is signaled. If the source operand of a conversion to a system-specific integer type is a NaN, infinity, or out-of-range for that format, then invalid is signaled (unless the type has an appropriate representation for the exceptional result). NaNs, infinities, and

out-of-range values are stored in a two's-complement integer format as the extreme negative value (for example, in the 16-bit integer format, as -32768).

Note that IEEE rounding into integral formats differs from most common rounding functions on halfway cases. With the default rounding direction (to nearest), conversions to comp or to a system-specific integer type will round 0.5 to 0, 1.5 to 2, 2.5 to 2, and 3.5 to 4, rounding to even on halfway cases. (Rounding is discussed in detail in Section 8.)

4.3 Conversions between Binary and Decimal

The IEEE Standard for binary floating-point arithmetic specifies the set of numerical values representable within each floating-point format. It is important to recognize that binary storage formats can exactly represent the fractional part of decimal numbers in only a few cases; in all other cases, the representation will be approximate. For example, 0.5_{10}, or $1/2_{10}$, can be represented exactly as 0.1_2. On the other hand, 0.1_{10}, or $1/10_{10}$, is a repeating fraction in binary: $0.00011001100..._2$. Its closest representation in single is $0.00011001100110011001100110_2$, which is closer to 0.100000001490_{10} than to 0.100000000000_{10}.

As binary storage formats generally provide only close approximations to decimal values, it is important that conversions between the two types be as accurate as possible. Given a rounding direction, for every decimal value there is a best (correctly rounded) binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-decimal conversions should obtain this best value to reduce accumulated errors. Conversion routines in SANE implementations meet or exceed the stringent error bounds specified by the IEEE Standard. This means that although in extreme cases the conversions do not deliver the correctly rounded result, the result delivered is very nearly as good as the correctly rounded result. (See the IEEE Standard [10] for a more detailed description of error bounds.)

4.3.1 Conversions from Decimal Strings to SANE Types

Routines may be provided to convert numeric decimal strings to the SANE data types. These routines are provided for the convenience of those who do not wish to write their own parsers and scanners. Examples of acceptable input are

```
123     123.4E-12     -123.     .456     3e9     -0

-INF    Inf     NAN(12)          -NaN()          nan
```

The 12 in NAN(12) is a NaN code (see Section 8).

The accepted syntax is formally defined, using Backus-Naur form, in Table 3-1:

Table 4-1.
Syntax for String Conversions

```
<decimal number>     ::=  [{space | tab}] <left decimal>
<left decimal>       ::=  [+|-] <unsigned decimal>
<unsigned decimal>   ::=  <finite number> | <infinity> | <NAN>
<finite number>      ::=  <significand> [<exponent>]
<significand>        ::=  <integer> | <mixed>
<integer>            ::=  <digits> [.]
<digits>             ::=  {0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9}
<mixed>              ::=  [<digits>] . <digits>
<exponent>           ::=  E [+|-] <digits>
<infinity>           ::=  INF
<NAN>                ::=  NAN[([<digits>])]
```

Note: Square brackets enclose optional items, curly brackets enclose elements to be repeated at least once, and vertical bars separate alternative elements; letters that appear literally, like the *E* marking the exponent field, may be either upper or lower case.

4.3.2 Decform Records and Conversions from SANE types to Decimal Strings

Each conversion to a decimal string is controlled by a decform record, which contains two fields:

```
style  -- 16-bit integer (0 or 1)
digits -- 16-bit integer
```

Style equals 0 for floating and 1 for fixed. Digits gives the number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style (digits may be negative if the style is fixed). Decimal strings resulting from these conversions are always acceptable input for conversions from decimal strings to SANE types. Further formatting details are implementation dependent.

4.3.3 The Decimal Record Type

The decimal record type provides an intermediate unpacked form for programmers who wish to do their own parsing of numeric input or formatting of numeric output. The decimal record format has three fields:

```
sgn -- 16-bit integer (0 or 1)
exp -- 16-bit integer
sig -- string (maximum length is implementation-dependent)
```

The value represented is

$$(-1)^{sgn} * sig * 10^{exp}$$

when the length of sig is 18 or less. (Some implementations allow additional information in characters past the eighteenth.) Sig contains the integral decimal significand: the initial byte of sig (sig[0]) is the length byte, which gives the length of the ASCII string that is left-justified in the remaining bytes. Sgn is 0 for + and 1 for -. For example, if sgn = 1, exp = -3, and sig = '85' (sig[0] = 2, not shown), then the number represented is -0.085.

4.3.4 Conversions from Decimal Records to SANE Types

Conversions from the decimal record type handle any sig digit-string of length 18 or less (with an implicit decimal point at the right end). The following special cases apply:

- If sig[1] = '0' (zero), the decimal record is converted to zero. For example, a decimal record with sig = '0913' is converted to zero.
- If sig[1] = 'N', the decimal record is converted to a NaN. Except when the destination is of type comp (which has a unique NaN), the succeeding characters of sig are interpreted as a hex representation of the result significand: if fewer than 4 characters follow N then they are right justified in the high-order 15 bits of the field f illustrated under Formats in Section 2; if 4 or more characters follow N then they are left justified in the result's significand; if no characters, or only 0's, follow N, then the result NaN code is set to nanzero = 15 (hex).
- If sig[1] = 'I' and the destination is not of comp type, the decimal record is converted to an infinity. If the destination is of comp type, the decimal record is converted to a NaN and invalid is signaled.
- Other special cases produce undefined results.

4.3.5 Conversions from SANE Types to Decimal Records

Each conversion to a decimal record is controlled by a decform record (see above). All implementations allow at least 18 digits to be returned in sig. The implied decimal point is at the right end of sig, with exp set accordingly.

Zeroes, infinities, and NaNs are converted to decimal records with sig parts 0 (zero), I, and strings beginning with N, while exp is undefined. For NaNs, N may be followed by a hex representation of the input significand. The third and forth hex digits following N give the NaN code. For example, 'N0021000000000000' has NaN code 21 (hex).

When the number of digits specified in a decform record exceeds an implementation maximum (which is at least 18), the result is undefined.

A number may be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for sig is 18, then 10^{15} (which requires 16 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than 2 digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the SANE implementation) one of two results is returned: an implementation may return the most significant digits of the number in sig and set exp so that the decimal record contains a valid floating-style representation of the number; alternatively, an implementation may simply set the string sig to '?'. In any implementation, the test

```
(-exp <> decform.digits) or (sig[1] = '?')
```

determines whether a nonzero finite number is too large for the chosen fixed style.

4.4 Conversions between Decimal Formats

SANE implementations may provide conversions between decimal strings and decimal records.

4.4.1 Conversion from Decimal Strings to Decimal Records

This conversion routine is intended as an aid to programmers doing their own scanning. The routine is designed for use either with fixed strings or with strings being received (interactively) character by character. An integer argument on input gives the starting index into the string and on output is one greater than the index of the last character in the numeric substring just parsed. The longest possible numeric substring is parsed; if no numeric substring is recognized, then the index remains unchanged. Also, a Boolean argument is returned indicating that the input string, beginning at the input index, is a valid numeric string or a valid prefix of a numeric string. The accepted input for this conversion is the same as for conversions from decimal strings to SANE types (see above). Output is the same as for conversions from SANE types to decimal records (also above).

Examples

Input String	Index in	Index out	Output Value	Valid Prefix
12	1	3	12	TRUE
12E	1	3	12	TRUE
12E-	1	3	12	TRUE
12E-3	1	6	12E-3	TRUE
12E-x	1	3	12	FALSE
12E-3x	1	6	12E-3	FALSE
x12E-3	2	7	12E-3	TRUE
IN	1	1	UNDEFINED	TRUE
INF	1	4	INF	TRUE

4.4.2 Conversion from Decimal Records to Decimal Strings

This conversion is controlled by the style field of a decform record (the digits field is ignored). Input is the same as for conversions from decimal records to SANE types, and output formatting is the same as for conversions from SANE types to decimal strings. This conversion, actually a formatting operation, is exact and signals no exception.

5 Expression Evaluation

SANE arithmetic is extended-based. Arithmetic operations produce results with extended precision and extended range. For minimal loss of accuracy in more complicated computations, you should use extended temporary variables to store intermediate results.

5.1 Using Extended Temporaries

A programmer may use extended temporaries deliberately to reduce the effects of round-off error, overflow, and underflow on the final result.

Example 1

To compute the single-precision sum

$$S = X[1]*Y[1] + X[2]*Y[2] + \ldots + X[N]*Y[N]$$

where X and Y are arrays of type single, declare an extended variable XS and compute

```
XS := 0;
FOR I := 1 TO N DO
  XS := XS + X[I]*Y[I];          {extended-precision arithmetic }
S := XS;                         {deliver final result to single.}
```

Even when input and output values have only single precision, it may be very difficult to prove that single-precision arithmetic is sufficient for a given calculation. Using extended-precision arithmetic for intermediate values will often improve the accuracy of single-precision results more than virtuoso algorithms would. Likewise, using the extra range of the extended type for intermediate results may yield correct final results in the single type in cases when using the single type for intermediate results would cause an overflow or a catastrophic underflow. Extended-precision arithmetic is also useful for calculations involving double or comp variables: see Example 2.

5.2 Extended-Precision Expression Evaluation

High-level languages that support SANE evaluate all non-integer numeric expressions to extended precision, regardless of the types of the operands.

Example 2

If C is of type comp and MAXCOMP is the largest comp value, then the right-hand side of

```
C := (MAXCOMP + MAXCOMP) / 2
```

would be evaluated in extended to the exact result C = MAXCOMP, even though the intermediate result MAXCOMP + MAXCOMP exceeds the largest possible comp value.

5.3 Extended-Precision Expression Evaluation and the IEEE Standard

The IEEE Standard encourages extended-precision expression evaluation. Extended evaluation will on rare occasions produce results slightly different from those produced by other IEEE implementations that lack extended evaluation. Thus in a single-only IEEE implementation,

```
z := x + y
```

with x, y, and z all single, is evaluated in one single-precision operation, with at most one rounding error. Under extended evaluation, however, the addition x + y is performed in extended, then the result is coerced to the single precision of z, with at most two rounding errors. Both implementations conform to the standard.

The effect of a single- or double-only IEEE implementation can be obtained under SANE with rounding precision control, as described in Section 8.

6 Comparisons

SANE supports the usual numeric comparisons: less, less-or-equal, greater, greater-or-equal, equal, and not-equal. For real numbers, these comparisons behave according to the familiar ordering of real numbers.

SANE comparisons handle NaNs and infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any SANE values a and b, exactly one of the following is true:

```
a < b
a > b
a = b
a and b are unordered
```

Determination is made by the rule:

If x or y is a NaN, then x and y are unordered; otherwise, x and y are less, equal, or greater according to the ordering of the real numbers, with the understanding that +0 = -0 = real 0, and $-\infty$ < each real number < $+\infty$.

(Note that a NaN always compares unordered--even with itself.)

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the Pascal or BASIC expression x <= y is true if x is less than y or if x equal y, and is false if x is greater than y or if x and y are unordered. Note that the SANE not-equal relation means less, greater, or unordered--even if not-equal is written <>, as in Pascal and BASIC. High-level languages supporting SANE supplement the usual comparison operators with a function that takes two numeric arguments and returns the appropriate relation (less, equal, greater, or unordered). This function can be used to determine whether two numeric representations satisfy any combination of less, equal, greater, and unordered.

A high-level language comparison that involves a relational operator containing less or greater, but not unordered, signals invalid if the operands are unordered (that is, if either operand is a NaN). For example, in Pascal or BASIC if x or y is a quiet NaN then x < y, x <= y, x >= y, and x > y signal invalid, but x = y and x <> y (recall that <> contains unordered) do not. If a comparison operand is a signaling NaN, then invalid is always signaled, just as in arithmetic operations.

7 Infinities, NaNs, and Denormalized Numbers

In addition to the normalized numbers supported by most floating-point packages, IEEE floating-point arithmetic also supports infinities, NaNs, and denormalized numbers.

7.1 Infinities

An *infinity* is a special bit pattern that can arise in one of two ways:

- When a SANE operation should produce an exact mathematical infinity (such as 1/0), the result is an infinity bit pattern.
- When a SANE operation attempts to produce a number with magnitude too great for the number's intended floating-point storage format, the result may (depending on the current rounding direction) be an infinity bit pattern.

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The infinities, one positive (+INF) and one negative (-INF) , generally behave as suggested by the theory of limits. For example, 1 added to +INF yields +INF; -1 divided by +0 yields -INF; and 1 divided by -INF yields -0.

Each of the storage types single, double, and extended provides unique representations for +INF and -INF. The comp type has no representations for infinities. (An infinity moved to the comp type becomes the comp NaN.)

7.2 NaNs

When a SANE operation cannot produce a meaningful result, the operation delivers a special bit pattern called a *NaN* (Not-a-Number). For example, 0 divided by 0, +INF added to -INF, and sqrt(-1) yield NaNs. A NaN can occur in any of the SANE storage types (single, double, extended, and comp); but, generally, system-specific integer types have no representation for NaNs. NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN (that is, has the same NaN code). If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: *quiet NaNs*, the usual kind produced by floating-point operations; and *signaling NaNs*. When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and, if no halt occurs, a quiet NaN is the delivered result. Signaling NaNs could be used for uninitialized variables. They are not created by any SANE operations. The most significant bit of the field f illustrated under Formats in Section 2 is clear for quiet NaNs and set for signaling NaNs. The unique comp NaN generally behaves like a quiet NaN.

A NaN in a floating-point format has an associated NaN code that indicates the NaN's origin. (These are listed in Table 7-1). The NaN code is the 8th through 15th most significant bits of the field f illustrated in Section 2. The comp NaN is unique and has no NaN code.

Table 7-1.
SANE NaN Codes

Name	Dec	Hex	Meaning
NANSQRT	1	$01	Invalid square root, such as sqrt(-1)
NANADD	2	$02	Invalid addition, such as (+INF) - (+INF)
NANDIV	4	$04	Invalid division, such as 0/0
NANMUL	8	$08	Invalid multiplication, such as 0 * INF
NANREM	9	$09	Invalid remainder or mod such as x rem 0
NANASCBIN	17	$11	Attempt to convert invalid ASCII string
NANCOMP	20	$14	Result of converting comp NaN to floating
NANZERO	21	$15	Attempt to create a NaN with a zero code
NANTRIG	33	$21	Invalid argument to trig routine
NANINVTRIG	34	$22	Invalid argument to inverse trig routine
NANLOG	36	$24	Invalid argument to log routine
NANPOWER	37	$25	Invalid argument to xi or xy routine
NANFINAN	38	$26	Invalid argument to financial function
NANINIT	255	$FF	Uninitialized storage (signaling NaN)

7.3 Denormalized Numbers

Whenever possible, floating-point numbers are *normalized* to keep the leading significand bit 1: this maximizes the resolution of the storage type. When a number is too small for a normalized representation, leading zeros are placed in the significand to produce a *denormalized* representation. A denormalized number is a nonzero number that is not normalized and whose exponent is the minimum exponent for the storage type.

Example

The sequence below shows how a single-precision value becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called *gradual underflow*.

$A_0 = 1.100\ 1100\ 1100\ 1100\ 1100\ 1101 * 2^{-126} \approx 0.1_{10} * 2^{-122}$

$A_1 = A_0/2 = 0.110\ 0110\ 0110\ 0110\ 0110\ 0110 * 2^{-126}$ (underflow)

$A_2 = A_1/2 = 0.011\ 0011\ 0011\ 0011\ 0011\ 0011 * 2^{-126}$

$A_3 = A_2/2 = 0.001\ 1001\ 1001\ 1001\ 1001\ 1010 * 2^{-126}$ (underflow)

. . . . . .

$A_{22} = A_{21}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0011 * 2^{-126}$

$A_{23} = A_{22}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0010 * 2^{-126}$ (underflow)

$A_{24} = A_{23}/2 = 0.000\ 0000\ 0000\ 0000\ 0000\ 0001 * 2^{-126}$

$A_{25} = A_{24}/2 = 0.0$ (underflow)

$A_1 ... A_{24}$ are denormalized; A_{24} is the smallest positive denormalized number in single type.

7.3.1 Why Denormalized Numbers?

The use of denormalized numbers makes statements like the following true for all real numbers:

```
x - y = 0  if and only if  x = y
```

This statement is not true for most older systems of computer arithmetic, because they exclude denormalized numbers. For these systems, the smallest nonzero number is a normalized number with the minimum exponent; when the result of an operation is smaller than this smallest normalized number, the system delivers zero as the result. For such *flush-to-zero* systems, if x ≠ y but x - y is smaller than the smallest normalized number, then x - y = 0. IEEE systems do not have this defect, as x - y, although denormalized, is not zero.

(A few old programs that rely on premature flushing to zero may require modification to work properly under IEEE arithmetic. For example, some programs may test x - y = 0 to determine whether x is very near y.)

7.4 Inquiries: Class and Sign

Each valid representation in a SANE data type (single, double, comp, or extended) belongs to exactly one of these classes:

- Signaling NaN.
- Quiet NaN.
- Infinite.
- Zero.
- Normalized.
- Denormalized.

SANE implementations provide the user with the facility to determine easily the class and sign of any valid representation.

Environmental controls include the rounding direction, rounding precision, exception flags, and halt settings.

8 Environmental Control

8.1 Rounding Direction

The available rounding directions are:

- To-nearest.
- Upward.
- Downward.
- Toward-zero.

The rounding direction affects all conversions and arithmetic operations except comparison and remainder. Except for conversions between binary and decimal (described in Section 4), all operations are computed as if with infinite precision and range and then rounded to the destination format according to the current rounding direction. The rounding direction may be interrogated and set by the user.

The default rounding direction is to-nearest. In this direction the representable value nearest to the infinitely precise result is delivered; if the two nearest representable values are equally near, the one with least significant bit zero is delivered. Hence, halfway cases round to even when the destination is the comp or a system-specific integer type, and when the round-to-integer operation is used. If the magnitude of the infinitely precise result exceeds the format's largest value (by at least one half unit in the last place), then the corresponding signed infinity is delivered.

The other rounding directions are upward, downward, and toward-zero. When rounding upward, the result is the format's value (possibly INF) closest to and no less than the infinitely precise result. When rounding downward, the result is the format's value (possibly -INF) closest to and no greater than the infinitely precise result. When rounding toward zero, the result is the format's value closest to and no greater in magnitude than the infinitely precise result. To truncate a number to an integral value, use toward-zero rounding either with conversion into an integer format or with the round-to-integer operation.

8.2 Rounding Precision

Normally, SANE arithmetic computations produce results to extended precision and range. To facilitate simulations of arithmetic systems that are not extended-based, the IEEE Standard requires that the user be able to set the rounding precision to single or double. If the SANE user sets rounding precision to single (or double) then all arithmetic operations produce results that are correctly rounded and that overflow or underflow as if the destination were single (or double), even though results are typically delivered to extended formats. Conversions to double and extended formats are

affected if rounding precision is set to single, and conversions to extended formats are affected if rounding precision is set to double; conversions to decimal, comp, and system-specific integer types are not affected by the rounding precision. Rounding precision can be interrogated as well as set.

Setting rounding precision to single or double does not significantly enhance performance, and in some SANE implementations may hinder performance.

8.3 Exception Flags and Halts

SANE supports five exception flags with corresponding halt settings:

- Invalid-operation (or invalid, for short).
- Underflow.
- Overflow.
- Divide-by-zero.
- Inexact.

These exceptions are signaled when detected; and, if the corresponding halt is enabled, the SANE engine will jump to a user-specified location. (A high-level language need not pass on to its user the facility to set this location, but may halt the user's program). The user's program can examine or set individual exception flags and halts, and can save and get the entire environment (rounding direction, rounding precision, exception flags, and halt settings). Further details of the halt (trap) mechanism are SANE implementation specific.

8.3.1 Exceptions

The *invalid-operation* exception is signaled if an operand is invalid for the operation to be performed. The result is a quiet NaN, provided the destination format is single, double, extended, or comp. The invalid conditions are these:

- (addition or subtraction) magnitude subtraction of infinities, for example, (+INF) + (-INF).
- (multiplication) 0 * INF.
- (division) 0/0 or INF/INF.
- (remainder) x rem y, where y is zero or x is infinite.
- (square root) if the operand is less than zero.
- (conversion) to the comp format or to a system-specific integer format when excessive magnitude, infinity, or NaN precludes a faithful representation in that format (see Section 4 for details).
- (comparison) via predicates involving < or >, but not "unordered," when at least one operand is a NaN.
- Any operation on a signaling NaN except sign manipulations (negate, absolute-value, and copy-sign) and class and sign inquiries.

The *underflow* exception is signaled when a floating-point result is both tiny and inexact (and therefore, perhaps significantly less accurate than it would be if the exponent range were unbounded). A result is considered tiny if, before rounding, its magnitude is smaller than its format's smallest positive normalized number.

The *divide-by-zero* exception is signaled when a finite nonzero number is divided by zero. It is also signaled, in the more general case, when an operation on finite operands produces an exact infinite result: for example, logb (0) returns -INF and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

The *overflow* exception is signaled when a floating-point destination format's largest finite number is exceeded in magnitude by what would have been the rounded floating-point result were the exponent range unbounded. (Invalid, rather than overflow, flags the production of an out-of-range value for an integral destination format.)

The *inexact* exception is signaled if the rounded result of an operation is not identical to the mathematical (exact) result. Thus, inexact is always signaled in conjunction with overflow or underflow.

Valid operations on infinities are always exact and therefore signal no exceptions. Invalid operations on infinities are described above.

8.4 Managing Environmental Settings

The environmental settings in SANE are global and can be explicitly changed by the user. Thus all routines inherit these settings and are capable of changing them. Often special precautions must be taken because a routine requires certain environment settings, or because a routine's settings are not intended to propagate outside the routine.

Example 1

The subroutine below uses to-nearest rounding while not affecting its caller's rounding direction. (Examples in this section use Pascal syntax. SANE implementations in other languages have operations with equivalent functionality.)

```
                  - - -
var r: RoundDir;          { local storage for rounding direction }
                  - - -
begin
    r := GetRound;            { save caller's rounding direction }
    SetRound (TONEAREST);     {          set to-nearest rounding }
        - - -
    SetRound (r)          {  restore caller's rounding direction }
end;
```

Note that, if the subroutine is to be reentrant, then storage for the caller's environment must be local.

SANE implementations may provide two efficient functions for managing the environment as a whole: procedure-entry and procedure-exit.

The procedure-entry function returns the current environment (for saving in local storage) and sets the default environment: rounding direction to-nearest, rounding precision extended, and exception flags and halts clear.

Example 2

The following subroutine runs under the default environment while not affecting its caller's environment.

```
          - - -
  var e: Environment;              { local storage for environment }

          - - -
  begin
     ProcEntry (e);                { save caller's environment and }
                                   { set default environment       }
          - - -
       SetEnvironment (e)            { restore caller's environment  }
end;
```

The procedure-exit function facilitates writing subroutines which appear to their callers to be atomic operations (such as addition, sqrt, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which may be irrelevant or misleading. Procedure-exit, which takes a saved environment as arguments, does the following:

1. It temporarily saves the exception flags (raised by the subroutine).
2. It restores the environment received as argument.
3. It signals the temporarily saved exceptions. (If enabled, halts could occur at this step.)

Thus exceptions signaled between procedure-entry and procedure-exit are hidden from the calling program unless the exceptions remain raised when the procedure-exit function is called.

Example 3

The following function signals underflow if its result is denormal, and overflow if its result is infinite, but hides spurious exceptions occurring from internal computations.

```
function compres: double;
      - - -
var e: Environment;              { local storage for environment }
    c: NumClass;                 { for class inquiry             }
      - - -
begin {compres}
   ProcEntry (e);                { save caller's environment and }
                                 { set default environment -     }
                                 { now halts disabled            }
      - - -
   compres := result;                  { result to be returned }
   c := ClassD (result);               { class inquiry         }
   ClearXcps;           { clear possibly spurious exceptions   }

{ now raise specified exception flags:                          }
   if c = INFINITE then SetException (OVERFLOW, TRUE)
   else if c = DENORMALNUM then SetException (UNDERFLOW, TRUE);
    ProcExit (e)                 { restore caller's environment,    }
                                 { including any halt enables,  and }
                                 { then signal exceptions from      }
                                 { subroutine                       }
end {compres} ;
```

9 Auxiliary Procedures

SANE includes a set of special routines--

negate,
absolute value,
copy-sign,
next-after,
scalb,
logb,

--which are recommended in an appendix to the IEEE Standard as aids to programming.

9.1 Sign Manipulation

The sign manipulation operations change only the sign of their argument. Negate reverses the sign of its argument. Absolute-value makes the sign of its argument positive. Copy-sign takes two arguments and copies the sign of one of its arguments onto the sign of its other argument.

These operations are treated as nonarithmetic in the sense that they raise no exceptions: even signaling NaNs do not signal the invalid-operation exception.

9.2 Next-After Functions

The floating-point values representable in single, double, and extended formats constitute a finite set of real numbers. The next-after functions (one for each of these formats) generate the next representable neighbor in the proper format, given an initial value x and another value y indicating a direction from the initial value.

Each of the next-after functions takes two arguments, x and y:

nextsingle(x,y) (x and y are single)
nextdouble(x,y) (x and y are double)
nextextended(x,y) (x and y are extended)

As elsewhere, the names of the functions may vary with the implementation.

9.2.1 Special Cases for Next-After Functions

If the initial value and the direction value are equal, then the result is the initial value.

If the initial value is finite but the next representable number is infinite, then overflow and inexact are signaled.

If the next representable number lies strictly between -M and +M, where M is the smallest positive normalized number for that format, and if the arguments are not equal, then underflow and inexact are signaled.

9.3 Binary Scale and Log Functions

The scalb and logb functions are provided for manipulating binary exponents.

Scalb efficiently scales a given number (x) by a given integer power (n) of 2, returning $x * 2^n$.

Logb returns the binary exponent of its input argument as a signed integral value. When the input argument is denormalized, the exponent is determined as if the input argument had first been normalized.

9.3.1 Special Cases for Logb

If x is infinite, logb(x) returns +INF.

If x = 0, logb(x) returns -INF and signals divide-by-zero.

10 The Elementary Functions

SANE provides a number of basic mathematical functions, including logarithms, exponentials, two important financial functions, trigonometric functions, and a random number generator. These functions are computed using the basic SANE arithmetic heretofore described.

All of the elementary functions, except the random number generator, handle NaNs, overflow, and underflow appropriately. All signal inexact appropriately, except that the general exponential and the financial functions may conservatively signal inexact when determining exactness would be too costly.

10.1 Logarithm Functions

SANE provides three logarithm functions.

- base-2 logarithm : $\log_2(x)$
- base-e or natural logarithm : $\ln(x)$
- base-e logarithm of 1 plus argument : $\ln1(x)$

Ln1(x) accurately computes ln(1 + x). If the input argument x is small, such as an interest rate, the computation of ln1(x) is more accurate than the straightforward computation of ln(1 + x) by adding x to 1 and taking the natural logarithm of the result.

10.1.1 Special Cases for Logarithm Functions

If x = +INF, then $\log_2(x)$, ln(x), and ln1(x) return +INF. No exception is signaled.

If x = 0, then $\log_2(x)$ and ln(x) return -INF and signal divide-by-zero. Similarly, if x = -1, then ln1(x) returns -INF and signals divide-by-zero.

If x < 0, then $\log_2(x)$ and ln(x) return a NaN and signal invalid. Similarly, if x < -1, then ln1(x) returns a NaN and signals invalid.

10.2 Exponential Functions

SANE provides five exponential functions.

- base-2 exponential : 2^x
- base-e or natural exponential : e^x
- base-e exponential

minus 1	:	expl(x)
- integer exponential	:	x^i (i of integer type)
- general exponential	:	x^y

Expl(x) accurately computes $e^x - 1$. If the input argument x is small, such as an interest rate, then the computation of expl(x) is more accurate than the straightforward computation of $e^x - 1$ by exponentiation and subtraction.

10.2.1 Special Cases for 2^x, e^x, expl(x)

If x = +INF, then 2^x, e^x, and expl(x) return +INF. No exception is signaled.

If x = -INF, then 2^x and e^x return 0; and expl(x) returns -1. No exception is signaled.

10.2.2 Special Cases for x^i

If the integer exponent i equals 0 and x is not a NaN, then x^i returns 1. Note that with the integer exponential, $x^0 = 1$ even if x is zero or infinite.

If x is +0 and i is negative, then x^i returns +INF and signals divide-by-zero.

If x is -0 and i is negative, then x^i returns +INF if i is even, or -INF if i is odd: both cases signal divide-by-zero.

10.2.3 Special Cases for x^y

If x is +0 and y is negative, then the general exponential xy returns +INF and signals divide-by-zero.

If x is -0 and y is integral and negative, then x^y returns +INF if y is even, or -INF if y is odd; both cases signal divide-by-zero.

The general exponential x^y returns a NaN and signals invalid if

both x and y equal 0;

x is infinite and y equals 0;

x = 1 and y is infinite; or

x is -0 or less than 0 and y is nonintegral.

10.3 Financial Functions

SANE provides two functions, compound and annuity, that can be used to solve various financial, or time-value-of-money, problems.

10.3.1 Compound

The compound function computes

$$compound(r, n) = (1 + r)^n$$

where r is the interest rate and n is the number (perhaps nonintegral) of periods. When the rate r is small, compound gives a more accurate computation than does the straightforward computation of $(1 + r)^n$ by addition and exponentiation.

Compound is directly applicable to computation of present and future values:

$$PV = FV * (1 + r)^{(-n)} = \frac{PV}{compound(r, n)}$$

$$FV = PV * (1 + r)^n = PV * compound(r, n)$$

10.3.2 Special Cases for Compound(r,n)

If r = 0 and n is infinite, or if r = -1, then compound(r,n) returns a NaN and signals invalid.

If r = -1 and n < 0, then compound(r,n) returns +INF and signals divide-by-zero.

10.3.3 Annuity

The annuity function computes

$$annuity(r, n) = \frac{1 - (1 + r)^{(-n)}}{r}$$

where r is the interest rate and n is the number of periods. Annuity is more accurate than the straightforward computation of the expression above using basic arithmetic operations and exponentiation. The annuity function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV = PMT * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * annuity(r, n)$$

$$FV = PMT * \frac{(1 + r)^n - 1}{r}$$

$$= PMT * (1 + r)n * \frac{1 - (1 + r)^{(-n)}}{r}$$

$$= PMT * compound(r, n) * annuity(r, n)$$

where PMT is the amount of one periodic payment.

10.3.4 Special Cases for Annuity(r,n)

If r = 0, then annuity(r,n) computes the sum of 1 + 1 + ... + 1 over n periods, and therefore returns the value n and signals no exceptions (the value n corresponds to the limit as r approaches 0).

If r < -1, then annuity(r,n) returns a NaN and signals invalid.

If r = -1 and n > 0, then annuity(r,n) returns -INF and signals divide-by-zero.

10.4 Trigonometric Functions

SANE provides the basic trigonometric functions:

```
cosine      :  cos(x)

sine        :  sin(x)

tangent     :  tan(x)

arctangent  :  arctan(x)
```

The arguments for cosine, sine, and tangent and the results of arctangent are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Section 3) and the nearest extended-precision approximation of pi/2. Thus the cosine, sine, and tangent functions have periods slightly different from their mathematical counterparts and diverge from their counterparts when their arguments become large. Number results from arctangent lie between -pi/2 and pi/2.

The remaining trigonometric functions can be easily and efficiently computed from these four (see Appendix C).

10.4.1 Special Cases for sin(x), cos(x):

If x is infinite, then cos(x) and sin(x) return a NaN and signal invalid.

10.4.2 Special Cases for tan(x):

If x is the nearest extended approximation to ±pi/2, then tan(x) returns ±INF.

If x is infinite, then tan(x) returns a NaN and signals invalid.

10.4.3 Special Case for arctan(x):

If x = ±INF, then arctan(x) returns the nearest extended approximation to ±pi/2.

10.5 Random Number Generator

SANE provides a pseudorandom number generator, random. Random has one argument, passed by address. A sequence of (pseudo)random integral values r in the range

$$1 \le r \le 2^{31} - 2$$

can be generated by initializing an extended variable r to an integral value (the seed) in the above range and making repeated calls random (r); each call delivers in r the next random number in the sequence.

If seed values of r are nonintegral or outside the range

$$1 \le r \le 2^{31} - 2$$

then results are unspecified.

A pseudorandom rectangular distribution on the interval (0,1) can be obtained by dividing the results from random by

$$2^{31} - 1 = \text{scalb}(31, 1) - 1 .$$

Appendix A
Bibliography

1. Apple Computer, Inc. "Appendix A: The Transcend and Realmodes Units" and "Appendix E: Floating-Point Arithmetic," *Apple III Pascal Programmer's Manual,* Volume 2, pp. 2-9, 56-85.

   These appendixes describe the implementation of single-precision arithmetic in Apple III Pascal, which was based upon Draft 8.0 of the proposed Standard.

2. Apple Computer, Inc. *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units.*

   This manual describes the Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units. This was Apple's first full implementation of IEEE arithmetic.

3. Apple Computer, Inc. *Apple III Pascal Numerics Manual: A Guide to Using the Apple III Pascal SANE and Elems Units.*

   This manual, generalized from the Apple III manual (number 2 above), describes the Apple II and Apple III Pascal implementation of the Standard Apple Numeric Environment (SANE) through procedure calls to the SANE and Elems units.

4. Cody, W. J. "Analysis of Proposals for the Floating-Point Standard." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 63-68.

   This paper compares the several contending proposals presented to the Working Group.

5. Coonen, Jerome T. "An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic." *IEEE Computer,* Vol. 13, No. 1 January 1980.

   This paper is a forerunner to the work on the draft Standard.

6. Coonen, Jerome T. "Underflow and the Denormalized Numbers." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 75-87.

7. Coonen, Jerome T. "Accurate, Yet Economical Binary-Decimal Conversions." To appear in *ACM Transactions on Mathematical Software.*

8. Demmel, James. "The Effects of Underflow on Numerical Computation." To appear in *SIAM Journal on Scientific and Statistical Computing.*

   These papers examine one of the major features of the proposed Standard, gradual underflow, and show how problems of bounded exponent range can be handled through the use of denormalized values.

9. Fateman, Richard J. "High-Level Language Implications of the Proposed IEEE Floating-Point Standard." *ACM Transactions on Programming Languages and Systems,* Vol. 4, No. 2, April 1982, pp. 239-257.

   This paper describes the significance to high-level languages, especially FORTRAN, of various features of the IEEE proposed Standard.

10. Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Standard for Binary Floating-Point Arithmetic." Proposed to IEEE, 345 East 47th Street, New York, NY 10017.

    The implementation of SANE is based upon Draft 10.0 of this Standard.

11. Floating-Point Working Group 754 of the Microprocessor Standards Committee, IEEE Computer Society. "A Proposed Standard for Binary Floating-Point Arithmetic." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 51-62.

    This is Draft 8.0 of the proposed Standard, which was offered for public comment. The current Draft 10.0 is substantially simpler than this draft; for instance, warning mode and projective mode have been eliminated, and the definition of underflow has changed. However, the intent of the Standard is basically the same, and this paper includes some excellent introductory comments by David Stevenson, Chairman of the Floating-Point Working Group.

12. Hough, D. "Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic." *IEEE Computer,* Vol. 14, No. 3, March 1981, pp. 70-74.

    This paper is an excellent introduction to the floating-point environment provided by the proposed Standard, showing how it facilitates the implementation of robust numerical computations.

13. Kahan, W. "Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard," *Interval Mathematics 1980* (ed. K.E.L. Nickel). New York: Academic Press, New York, 1980, pp. 99-128.

    This paper shows how the proposed Standard facilitates interval arithmetic.

Appendix B
Glossary

application type: A data type used to store data for applications.

arithmetic type: A data type used to hold results of calculations inside the computer. The SANE arithmetic type, extended, has greater range and precision than the application types, in order to improve the mathematical properties of the application types.

binary floating-point number: A string of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and two raised to the power of its exponent.

comp type: A 64-bit application data type for storing integral values of up to 18- or 19-decimal-digit precision. It is used for accounting applications, among others.

denormalized number, or **denorm:** A nonzero binary floating-point number that is not normalized (that is, whose significand has a leading bit of zero) and whose exponent is the minimum exponent for the number's storage type.

double type: A 64-bit application data type for storing floating-point values of up to 15- or 16-decimal-digit precision. It is used for statistical and financial applications, among others.

environmental settings: The rounding direction and rounding precision, plus the exception flags and their respective halts.

exceptions: Special cases, specified by the IEEE Standard, in arithmetic operations. The exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

exception flag: Each exception has a flag that can be set, cleared and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

exponent: The part of a binary floating-point number that indicates the power to which two is raised in determining the value of the number. The wider the exponent field in a numeric type, the greater range it will handle.

extended type: An 80-bit arithmetic data type for storing floating-point values of up to 19- or 20-decimal-digit precision. SANE uses it to hold the results of arithmetic operations.

halt: Each exception has a halt-enable that can be set or cleared. When an exception is signaled and the corresponding halt is enabled, the SANE engine will transfer control to the address in a halt vector. A high-level language need not pass on to its user the facility to get the halt vector, but may halt the user's program. Halts remain set until explicitly cleared.

infinity: A special bit pattern produced when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

integer types: System types for integral values. Integer types typically use 16- or 32-bit two's complement integers. Integer types are not SANE types but are available to SANE users.

integral value: A value in a SANE type that is exactly equal to a mathematical integer: ..., -2, -1, 0, 1, 2, ...

NaN (Not a Number): A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs can also be used for uninitialized storage. NaNs propagate through arithmetic operations.

normalized number: A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1.

quiet NaN: A NaN that propagates through arithmetic operations without signaling an exception (and hence without halting a program).

rounding direction: When the result of an arithmetic operation cannot be represented exactly in a SANE type, the computer must decide how to round the result. Under SANE, the computer resolves rounding decisions in one of four directions, chosen by the user: to-nearest (the default), upward, downward, and toward-zero.

sign bit: The bit of a single, double, comp, or extended number that indicates the number's sign: 0 indicates a positive number; 1, a negative number.

signaling NaN: A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No SANE operation creates signaling NaNs.

significand: The part of a binary floating-point number that indicates where the number falls between two successive powers of two. The wider the significand field in a numeric type, the more resolution it will have.

single type: A 32-bit application data type for storing floating-point values of up to 7- or 8-decimal-digit precision. It is used for engineering applications, among others.

Appendix C
Other Elementary Functions

High quality transcendental functions which are not part of the Standard Apple Numeric Environment (SANE) can be constructed from the functions which SANE provides. Some common functions are provided below in pseudo-code. It should be relatively easy to adapt them for your use.

These functions are based on algorithms developed by Professor William Kahan, University of California at Berkeley. They are robust and accurate. The constant C is 2^{-33} = scalb (-33,1). It is chosen to be nearly the largest value for which $(1 - C^2)$ rounds to 1. All variables are extended.

Exception Handling

Unlike the SANE elementary functions, these functions do not provide complete handling of special-cases and exceptions. The most troublesome exceptions can be correctly handled if you:

- Begin each function with a call to procedure-entry.
- Clear the spurious exceptions indicated.
- End each function with a call to procedure-exit (see Section 8).

Functions

Secant

```
sec(x) <--- 1 / cos(x)
```

CoSecant

```
csc(x) <--- 1 / sin(x)
```

CoTangent

```
cot(x) <--- 1 / tan(x)
```

ArcSine

```
y <--- |x|
If y ≥ 0.3 then begin
                  y <--- Atan (x/sqrt ((1-x)*(1+x)))
                  spurious divide-by-zero may arise
                end
else if y ≥ C then y <--- Atan (x / (sqrt (1 - x^2))
               else y <--- x
arcsin(x) <--- y
```

ArcCosine

```
arccos(x) <--- 2 * Atan (sqrt ((1-x)/(1+x)) )
spurious divide-by-zero may arise
```

Sinh

```
y <--- |x|
If y ≥ C then begin
                y <--- expm1(y)
                y <--- 0.5 * (y + y/(1+y))
              end
copy the sign of x onto y
sinh(x) <--- y
```

Cosh

```
y <--- exp(|x|)
cosh(x) <--- 0.5 * y  +  0.25 / (0.5 * y)
```

Tanh

```
y <--- |x|
If y ≥ C then begin
                y <--- expm1(-2*y)
                y <--- -y/(2 + y)
              end
copy the sign of x onto y
tanh(x) <--- y
```

```
ArcSinh

    y <--- |x|
    If y ≥ C then begin
                    y <--- ln1 (y + y / (1/y + sqrt(1 + (1/y)^2) ))
                    spurious underflow may arise
                  end
    copy the sign of x onto y
    asinh(x) <--- y

ArcCosh

    y <--- |x|
    acosh(x) <--- ln1 ( (sqrt (y-1)) * (sqrt (y-1) + sqrt (y+1)) )

ArcTanh

    y <--- |x|
    If y ≥ C then y <--- ln1 (2*y/(1 - y)) / 2
    copy the sign of x onto y
    atanh(x) <--- y
```

The 68000 Assembly-Language SANE Engine

Contents

The 68000 Assembly-Language SANE Engine

1 Introduction

The purpose of the software package described in this manual is to provide the features of the Standard Apple Numeric Environment (SANE) to assembly-language programmers on Apple's 68000-based systems. SANE--described in detail in *The Standard Apple Numeric Environment* in this binder--fully supports the IEEE Standard (754) for Binary Floating-Point Arithmetic; it augments the Standard to provide greater utility for applications in accounting, finance, science, and engineering. The IEEE Standard and SANE offer a combination of quality, predictability, and portability heretofore unknown for numerical software.

A functionally equivalent 6502 assembly-language SANE engine is available for Apple's 6502-based systems. Thus numerical algorithms coded in assembly language for an Apple 68000-based system can be readily recoded for an Apple 6502-based system. Suggested macros for accessing the 6502 and 68000 engines have been chosen to further facilitate algorithm portability.

This manual describes the use of the 68000 Assembly-Language SANE engine, but does not describe SANE itself. For example, this manual explains how to call the SANE remainder function from 68000 assembly language but does not discuss what this function does. See *The Standard Apple Numeric Environment T* for information about the semantics of SANE.

See Appendix A for information about accessing the 68000 SANE engine from the Apple 68000-based systems.

2 Basics

The following code illustrates a typical invocation of the SANE engine, FP68K.

```
PEA     A_ADR   ; Push address of A (single format)
PEA     B_ADR   ; Push address of B (extended format)
FSUBS           ; Floating-point SUBtract Single: B <— B - A
```

FSUBS is an assembly-language macro taken from the file listed in Appendix B. The form of the operation in the example (B <-- B - A, where A is a numeric type and B is extended) is similar to the forms for most FP68K operations. Also, this example is typical of SANE engine calls because operands are passed to FP68K by pushing the addresses of the operands onto the stack prior to the call. Details of SANE engine access are given later in this section.

The SANE elementary functions are provided in Elems68K. Access to Elems68K is similar to access to FP86K; details are given in Section 9.

2.1 Operation Forms

The example above illustrates the form of an FP68K binary operation. Forms for other FP68K operations are described in this section. Examples and further details are given in subsequent sections.

2.1.1 Arithmetic and Auxiliary Operations

Most numeric operations are either unary (one operand), like square root and negation, or binary (two operands), like addition and multiplication.

The 68000 assembly-language SANE engine, FP68K, provides unary operations in a one-address form:

```
DST <-- <op> DST          ... for example, B <-- sqrt(B)
```

The operation <op> is applied to (or operates on) the operand DST and the result is returned to DST, overwriting the previous value. DST is called the destination operand.

FP68K provides binary operations in a two-address form:

```
DST <-- DST <op> SRC      ... for example, B <-- B / A
```

The operation <op> is applied to the operands DST and SRC and the result is returned to DST, overwriting the previous value. SRC is called the source operand.

In order to store the result of an operation (unary or binary), the location of the operand DST must be known to FP68K, so DST is passed by address to FP68K. In general all operands, source and destination, are passed by address to FP68K.

For most operations the storage format for a source operand (SRC) can be one of the SANE numeric formats (single, double, extended, or comp). To support the extended-based SANE arithmetic, a destination operand (DST) must be in the extended format.

The forms for the copy-sign next-after functions are unusual and will be discussed in Section 4.

2.1.2 Conversions

FP68K provides conversions between the extended format and other SANE formats, between extended and 16- or 32-bit integers, and between extended and decimal records. Conversions between binary formats (single, double, extended, comp, and integer) and conversions from decimal to binary have the form

```
DST <-- SRC
```

Conversions from binary to decimal have the form

```
DST <-- SRC according to SRC2
```

where SRC2 is a DecForm record specifying the decimal format for the conversion of SRC to DST.

2.1.3 Comparisons

Comparisons have the form

```
<relation> <-- SRC, DST
```

where DST is extended and SRC is single, double, comp, or extended, and where <relation> is less, equal, greater, or unordered according as

```
DST <relation> SRC
```

Here the result <relation> is indicated by setting the 68000 CCR flags.

2.1.4 Other Operations

FP68K provides inquiries for determining the class and sign of an operand and operations for accessing the floating-point environment word and the halt address. Forms for these operations vary and will be given as the operations are introduced.

2.2 External Access

The SANE engine, FP68K, is reentrant, position-independent code, which may be shared in multi-process environments. It is accessed through one entry point, labeled FP68K. Each user process has a static state area consisting of one word of mode bits and error flags, and a two-word halt vector. The package allows for different access to the state word in single and multi-process environments.

The package preserves all 68000 registers across invocations, except that REMAINDER modifies D0. The package modifies the 68000 CCR flags. Except for binary-decimal conversions, it uses little more stack area than is required to save the sixteen 32-bit 68000 registers. Since the binary-decimal

conversions themselves call the package (to perform multiplies and divides), they use about twice the stack space of the regular operations.

The access constraints described in this section also apply to Elems68K.

2.3 Calling Sequence

A typical invocation of the engine consists of a sequence of PEA's to push operand addresses followed by one of the Appendix B macros:

```
PEA     <source address>
PEA     <destination address>
<FOPMACRO>
```

PEA's for source operands always precede those for destination operands.

<FOPMACRO> represents a typical operation macro defined as

```
MOVE.W  <opword>,-(SP)          ; Push op code.
JSRFP
```

The macro JSRFP in turn generates a call to FP68K; for Macintosh, it expands to an A-line trap, while for Lisa it expands to an intrinsic unit subroutine call

```
JSR     FP68K
```

2.3.1 The Opword

The opword is the logical OR of a operand format code and an operation code.

The operand format code specifies the format (extended, double, single, integer, or comp) of one of the operands. The operand format code typically gives the format for the source operand (SRC). At most one operand format need be specified, since other operands' formats are implied.

The operation code specifies the operation to be performed by FP68K.

Opwords are listed in Appendix C; operand format codes and operation codes are listed in Appendix B.

Example

The format code for single is 0200 (hex). The operation code for divide is 0006 (hex). Hence the opword 0206 (hex) indicates divide by a value of type single.

2.3.2 Assembly-Language Macros

The macro file in Appendix B provides macros for

```
MOVE.W  <opword>,-(SP)
JSRFP
```

for most common <opword> calls to FP68K.

Example 1

To add a single-format operand A to an extended-format operand B, simply write:

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FADDS           ; Floating-point ADD Single: B <— B + A
```

Example 2

Compute B <-- sqrt(A), where A and B are extended. The value of A should be preserved.

```
PEA     A_ADR   ; Push address of A
PEA     B_ADR   ; Push address of B
FX2X            ; Floating-point eXtended to eXtended: B <— A
PEA     B_ADR   ; Push address of B
FSQRTX          ; Floating SQuare RooT eXtended: B <— sqrt(B)
```

Example 3

Compute C <-- A - B, where A, B, and C are in the double format. Since destinations are extended, a temporary extended variable T is required.

```
PEA     A_ADR   ; Push address of A
PEA     T_ADR   ; Push address of 10-byte temporary variable
FD2X            ; Fl-pt convert Double to eXtended: T <— A
PEA     B_ADR   ; Push address of B
PEA     T_ADR   ; Push address of temporary
FSUBD           ; Fl-pt SUBtract Double: T <— T - B
PEA     T_ADR   ; Push address of temporary
PEA     C_ADR   ; Push address of C
FX2D            ; Fl-pt convert eXtended to Double: C <—
```

2.4 Arithmetic Abuse

FP68K is designed to be as robust as possible, but it is not bulletproof. Passing the wrong number of operands to the engine will damage the stack. Using UNDEFINED opword parameters or passing incorrect addresses will produce undefined results.

3 Data Types

FP68K fully supports the SANE data types

```
single    -- 32-bit floating-point
double    -- 64-bit floating-point
comp      -- 64-bit integer
extended  -- 80-bit floating-point
```

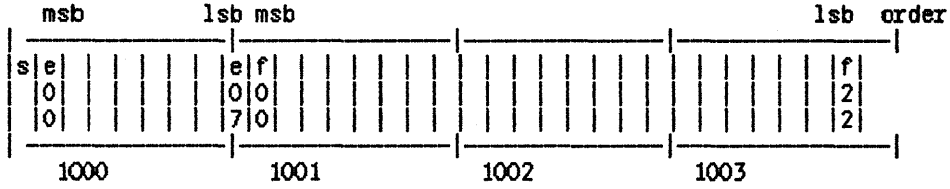
and the 68000-specific types

```
integer   -- 16-bit two's complement integer
longint   -- 32-bit two's complement integer
```

The 68000 engine uses the convention that least-significant bytes are stored in high memory. For example, let us take a variable of type single with bits

```
s               -- sign
e0 ... e7       -- exponent (msb...lsb)
f0 ... f22      -- significand fraction (msb...lsb)
```

The logical structure of this four-byte variable is shown below:



If this variable is assigned the address 1000, then its bits are distributed to the locations 1000 to 1003 as shown. The other SANE formats (see Section 2 in *The Standard Apple Numeric Environment*) are represented in memory in similar fashion.

4 Arithmetic Operations and Auxiliary Routines

The operations covered in this section follow the access schemes described in Section 2.

```
unary operations:  DST <-- <op> DST          (one-address form)

    PEA     <DST address>
    <FOPMACRO>

binary operations:  DST <-- DST <op> SRC     (two-address form)

    PEA     <SRC address>
    PEA     <DST address>
    <FOPMACRO>
```

The destination operand (DST) for these operations is passed by address and is generally in the extended format. The source operand (SRC) is also passed by address and may be single, double, comp, or extended. Some operations are distinguished by requiring some specific type for SRC, by using a nonextended destination, or by returning auxiliary information in the D0 register and in the processor CCR status bits. In this section, operations so distinguished are noted. The examples employ the macros in Appendix B.

4.1 Add, Subtract, Multiply, and Divide

These are binary operations and follow the two-address form.

Example

B <-- B / A , where A is double and B is extended:

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FDIVD           ; divide with source operand of type double
```

4.2 Square Root

This is a unary operation and follows the one-address form.

Example

B <-- sqrt(B) , where B is extended.

```
PEA     B_ADR   ; push address of B
FSQRTX          ; square root (operand is always extended)
```

4.3 Round-to-Integer, Truncate-to-Integer

These are unary operations and follow the one-address form.

Round-to-integer rounds (according to the current rounding direction) to an integral value in the extended format. Truncate-to-integer rounds toward zero (regardless of the current rounding direction) to an integral value in the extended format. The calling sequence is the usual one for unary operators, illustrated above for square root.

4.4 Remainder

This is a binary operation and follows the two-address form.

Remainder returns auxiliary information: the low-order integer quotient (between -127 and +127) in D0.W. The high half of D0.L is undefined. This intrusion into the register file is extremely valuable in argument reduction--the principal use of the remainder function. The state of D0 after an invalid remainder is undefined.

Example

B <-- B rem A , where A is single and B is extended.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FREMS           ; remainder with source operand of type single
```

4.5 Logb, Scalb

Logb is a unary operation and follows the one-address form.

Scalb is a binary operation and follows the two-address form. Its source operand is a 16-bit integer.

Example

B <-- B * 2^I, where B is extended.

```
PEA     I_ADR   ; push address of I
PEA     B_ADR   ; push address of B
FSCALBX         ; scalb
```

4.6 Negate, Absolute Value, Copy-Sign

Negate and absolute value are unary operations and follow the one-address form.

Copy-sign uses the calling sequence

```
PEA     <SRC address>
PEA     <DST address>
FCPYSGNX
```

to copy the sign of DST onto the sign of SRC. Note that copy-sign differs from most two-address operations in that it changes the SRC value rather than the DST value. The formats of the operands for FCPYSGNX can be single, double, or extended. (For efficiency, the 68000 assembly-language programmer should copy signs directly rather than call FP68K.)

Example

Copy the sign of B (single, double, or extended) into the sign of A (single, double, or extended).

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FCPYSGNX        ; copy-sign
```

4.7 Next-After

The next-after operations use the calling sequence

```
PEA     <SRC address>
PEA     <DST address>
<next-after macro>
```

to effect SRC <-- next value, in the format indicated by the macro, after SRC in the dirction of DST. Next-after operations differ from most two-address operations in that they change SRC values rather than DST values. Both source and destination operands must be of the same floating-point type (single, double, or extended).

Example

A <-- next-after(A) in the direction of B, where A and B are double (so *next-after* means *next-double-after*).

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FNEXTD          ; next-after in double format
```

5 Conversions

This section discusses conversions between binary formats and conversions between binary and decimal formats.

5.1 Conversions Between Binary Formats

FP68K provides conversions between the extended type and the SANE types single, double, and comp, as well as the 16- and 32-bit integer types.

5.1.1 Conversions to Extended

FP68K provides conversions of a source, of type single, double, comp, extended, or integer, to an extended destination.

```
                          single
                          double
extended          <--     comp
                          extended
                          integer
```

All operands, even integer ones, are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B, where A is of type comp and B is extended.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FC2X            ; convert comp to extended
```

5.1.2 Conversions from Extended

FP68K provides conversions of an extended source to a destination of type single, double, comp, extended, or integer.

```
single
double
comp              <--     extended
extended
integer
```

(Conversion to a narrower format may alter values.) Contrary to the usual scheme the destination for these conversions need not be of type extended. All operands are passed by address. The following example illustrates the calling sequence.

Example

Convert A to B where A is extended and B is double.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FX2D            ; convert extended to double
```

5.2 Binary-Decimal Conversions

FP68K provides conversions between the binary types (single, double, comp, extended, and integer) and the decimal record type.

Decimal records and decform records (used to specify the form of decimal representations) are described in Section 4 of *The Standard Apple Numeric Environment*. For FP68K, the maximum length of the sig digits field of a decimal record is 20. (The value 20 is specific to this implementation: algorithms intended to port to other SANE implementations should use no more than 18 digits in sig.)

5.2.1 Binary to Decimal

The calling sequence for a conversion from a binary format to a decimal record passes the address of a decform record, the address of a binary source operand, and the address of a decimal-record destination. The maximum number of significant digits that will be returned is 19.

Example

Convert a comp-format value A to a decimal record D according to the decform record F.

```
PEA     F_ADR   ; push address of F
PEA     A_ADR   ; push address of A
PEA     D_ADR   ; push address of D
FC2DEC          ; convert comp to decimal
```

Fixed-Format "Overflow"

If a number is too large for a chosen fixed style, then FP68K returns the string '?' in the sig field of the decimal record.

5.2.2 Decimal to Binary

The calling sequence for a conversion from decimal to binary passes the address of a decimal-record source operand and the address of a binary destination operand.

The maximum number of digits in sig is 19. If the length of sig is 20, then sig represents its first 19 digits plus one or more additional nonzero digits after the 19th. The exponent corresponds to the 19-digit integer represented by the first 19 digits of sig.

Example

Convert the decimal record D to a double-format value B.

```
PEA     D_ADR   ; push address of D
PEA     B_ADR   ; push address of B
FDEC2D          ; convert decimal to double
```

Techniques for Extreme Accuracy

The following techniques apply to FP68K; other SANE implementations require other techniques.

For maximum accuracy, insert or delete trailing zeros for the sig field of a decimal record in order to minimize the magnitude of the exp field. For example, for 1.0E60 set sig to '10000000000000000000000000000000' (17 zeros) and exp to 43, and for 300E-43 set sig to '3' and exp to -41.

If you are writing a parser and must handle a number with more than 19 significant digits, follow these rules:

- Place the implicit decimal point to the right of the 19 most significant digits.
- If any of the discarded digits to the right of the implicit decimal point are nonzero, then concatenate the digit '1' to sig.

6 Comparisons and Inquiries

6.1 Comparisons

FP68K offers two comparison operations: FCPX (which signals invalid if its operands compare unordered) and FCMP (which does not). Each compares a source operand (which may be single, double, extended, or comp) with a destination operand (which must be extended). The result of a comparison is the relation (less, greater, equal, or unordered) for which

```
DST <relation> SRC
```

is true. The result is delivered in the X, N, Z, V, and C status bits:

<relation>	Status bits X N Z V C
greater	0 0 0 0 0
less	1 1 0 0 1
equal	0 0 1 0 0
unordered	0 0 0 1 0

These status bit encodings reflect that floating-point comparisons have four possible results, unlike the more familiar integer comparisons with three possible results. It's not necessary to learn these encodings, however; simply use the FBxxx series of macros for branching after FCMP and FCPX.

FCMP and FCPX are both provided to facilitate implementation of relational operators defined by higher level languages that do not contemplate unordered comparisons. The IEEE standard specifies that the invalid exception shall be signalled whenever necessary to alert users of such languages that an unordered comparison may have adversely affected their program's logic.

Example 1

Test B <= A, where A is single and B is extended; if TRUE branch to LOC; signal if unordered.

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FCPXS           ; compare using source of type single,
                ; signal invalid if unordered
FBLE    LOC     ; branch if B <= A
```

Example 2

Test B not-equal A, where A is double and B is extended; if TRUE branch to LOC. (Note that not-equal is equivalent to less, greater, or unordered, so invalid should not be signaled on unordered.)

```
PEA     A_ADR   ; push address of A
PEA     B_ADR   ; push address of B
FCMPD           ; compare using source of type double,
                ; do not signal invalid if unordered
FBNE    LOC     ; branch if B not-equal A
```

6.2 Inquiries

The classify operation provides both class and sign inquiries. This operation takes one source operand (single, double, or extended), which is passed by address, and places the result in a 16-bit integer destination.

The sign of the result is the sign of the source; the magnitude of the result is

```
1       signaling NaN
2       quiet NaN
3       infinite
4       zero
5       normal
6       denormal
```

Example

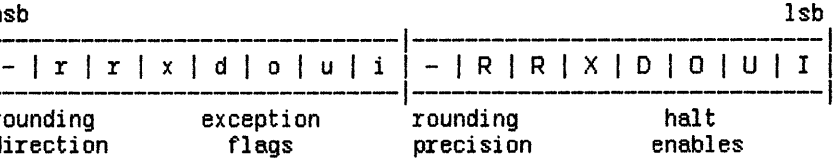
Set C to sign and class of A.

```
PEA     A_ADR   ; push address of A
PEA     C_ADR   ; push address of result
FCLASSS         ; classify single
```

7 Environmental Control

7.1 The Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:




```
rounding direction, bits 6000                rr
        0000 -- to-nearest
        2000 -- upward
        4000 -- downward
        6000 -- toward-zero

exception flags, bits 1F00
        0100 -- invalid                      i
        0200 -- underflow                    u
        0400 -- overflow                     o
        0800 -- division-by-zero             d
        1000 -- inexact                      x

rounding precision, bits 0060                RR
        0000 -- extended
        0020 -- double
        0040 -- single
        0060 -- UNDEFINED

halt enabled, bits 001F
        0001 -- invalid                      I
        0002 -- underflow                    U
        0004 -- overflow                     O
        0008 -- division-by-zero             D
        0010 -- inexact                      X
```

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

Example

With rounding toward-zero, inexact and underflow exception flags raised, extended rounding precision, and halt on invalid, overflow, and division-by-zero, the most significant byte of the environment is 72 and the least significant byte is 0D.

Access to the environment is via the operations get-environment, set-environment, test-exception, set-exception, procedure-entry, and procedure-exit.

7.2 Get-Environment and Set-Environment

Get-Environment takes one input operand: the address of a 16-bit integer destination. The environment word is returned in the destination.

Set-Environment has one input operand: the address of a 16-bit integer, which is to be interpreted as an environment word.

Example

Set rounding direction to toward-zero.

```
PEA     A_ADR
FGETENV
MOVE.W  (A0),D0         ; D0 gets environment
OR.W    #$6000,D0       ; set rounding toward-zero
MOVE.W  D0,(A0)         ; restore A
PEA     A_ADR
FSETENV
```

7.3 Test-Exception and Set-Exception

Test-exception has one integer destination operand, which contains the hex values

```
01 -- invalid
02 -- underflow
04 -- overflow
08 -- divide-by-zero
10 -- inexact
```

If the exception flag is set for the corresponding bit in the operand, then test-exception sets the destination to $100, otherwise, to zero.

Set-exception takes one integer source operand, which encodes an exception in the manner described above for test-exception. Set-exception stimulates the exception indicated in the operand.

7.4 Procedure-Entry and Procedure-Exit

Procedure-entry saves the current floating-point environment (16-bit integer) at the address passed as the sole operand, and sets the operative environment to the default state.

Procedure-exit saves (temporarily) the exception flags, sets the environment passed as the sole operand, and then stimulates the saved exceptions.

Example

Here is a procedure that appears to its callers as an atomic operation.

```
ATOMICPROC
        PEA     E_ADR   ; push address to store environment
        FPROCENTRY      ; procedure entry

        ...body of routine...

        PEA     E_ADR   ; push address of environment
        FPROCEXIT       ; procedure exit
RTS
```

8 Halts

FP68K provides the facility to transfer program control when selected floating-point exceptions occur. Since this facility will be used to implement halts in high-level languages, we refer to it as a halting mechanism. The assembly-language programmer can write a 'halt handler' routine to cause special actions for floating-point exceptions. The FP68K halting mechanism differs from the traps that are an optional part of the IEEE Standard.

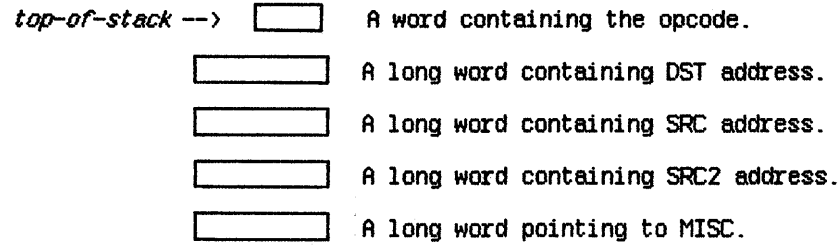
8.1 Conditions for a Halt

Any floating-point exception can, under the appropriate conditions, trigger a halt. The halt for a particular exception is enabled when the user has set the halt-enable bit corresponding to that exception.

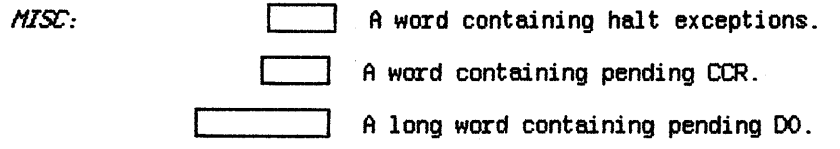
8.2 The Halt Mechanism

If the halt for a given exception is enabled, FP68K does these things when that exception occurs:

1. FP68K returns the same result to the destination address that it would return if the halt were not enabled.
2. It sets up the following stack frame:



MISC is a record consisting of:



The first word of MISC contains in its five low-order bits the AND of the halt-enable bits with the exceptions that occurred in the operation just completing. If halts were not enabled, then (upon return from FP68K) CCR and D0 would have the values given in MISC.

3. It passes control by JSR through the halt vector previously set by FSETHV, pushing another long word containing a return address in FP68K. If execution is to continue, the halt procedure must clear eighteen bytes from the stack to remove the opword and the DST, SRC, SRC2, and MISC addresses.

Set-halt-vector has one input operand: the address of a 32-bit integer, which is interpreted as the halt vector (that is, the address to jump to in case a halt occurs).

Get-halt-vector has one input operand: the address of a 32-bit integer, which receives the halt vector.

8.3 Using the Halt Mechanism

This example illustrates the use of the halting mechanism. The user must set the halt vector to the starting address of a halt handler routine. This particular halt handler returns control to FP68K which will continue as if no halt had occurred, returning to the next instruction in the user's program.

```
        LEA     HROUTINE,A0     ; A0 gets address of halt routine
        MOVE.L  A0,H_ADR        ; H_ADR gets same
        PEA     H_ADR           ;
        FSETHV                  ; set halt vector to HROUTINE
        ...
        PEA                     ; floating-point operand here
        <FOPMACRO>              ; a floating-point call here
        . . .
HROUTINE                        ; called by FP68K
        MOVE.L  (SP)+,A0        ; A0 saves return address in FP68K
        ADD.L   #18,SP          ; increment stack past arguments
        JMP     (A0)            ; return to FP68K
```

The FP68K halt machanism is designed so that a halt procedure may be written in Lisa Pascal. This is the form of a Pascal equivalent to HROUTINE:

```
type    miscrec = record
                  halterrors : integer ;
                  ccrpending : integer ;
                  DOpending : longint ;
                end {record} ;

procedure haltroutine
        ( var misc : miscrec ;
          src2, src, dst : longint ;
          opcode : integer ) ;

begin {haltroutine}
end {haltroutine} ;
```

Like HROUTINE, haltroutine merely continues execution as if no halt had occurred.

9 Elementary Functions

The elementary functions that are specified by the Standard Apple Numeric Environment are made available to the 68000 assembly-language programmer in ELEMS68K. Also included are two functions that compute $\log_2(1+x)$ and $2^x - 1$ accurately. ELEMS68K calls the SANE engine (FP68K) for its basic arithmetic. The access schemes for FP68K (described in Section 2) and ELEMS68K are similar. Opwords and sample macros are included at the end of the file listed in Appendix B. (These macros will be used freely in the examples below.)

9.1 One-Argument Functions

The SANE elementary functions $\log_2(x)$, $\ln(x)$, $\ln1(x) = \ln(1+x)$, 2^x, e^x, $\exp1(x) = e^x - 1$, $\cos(x)$, $\sin(x)$, $\tan(x)$, $\text{atan}(x)$, and $\text{random}(x)$, together with $\log21(x) = \log_2(1+x)$ and $\exp21(x) = 2^x - 1$, each have one extended argument, passed by address. These functions use the one-address calling sequence

```
PEA     DST
<EOPMACRO>
```

to effect

```
DST <-- <op> DST
```

<EOPMACRO> is one of the macros in appendix B that generate code to push an op word and invoke ELEMS68K. This follows the FP68K access scheme for unary operations, such as square root and negate.

Example

B <-- sin(B), where B is of extended type.

```
PEA     B_ADR   ; push address of B
FSINX           ; B <— sin(B)
```

9.2 Two-Argument Functions

General exponentiation (x^y) has two extended arguments, both passed by address. The result is returned in x.

Integer exponentiation (x^i) also has two arguments. The extended argument x, passed by address, receives the result. The 16-bit integer argument i is also passed by address.

Both exponentiation functions use the calling sequence for binary operations

```
PEA     SRC address     ; push exponent address first
PEA     DST address     ; push base address second
<EOPMACRO>
```

to effect

```
DST <-- DST^SRC
```

Example

B <-- B^K, where the type of B is extended.

```
PEA     K_ADR   ; push address of K
PEA     B_ADR   ; push address of B
FXPWRI          ; integer exponentiation
```

9.3 Three-Argument Functions

Compound and annuity use the calling sequence

```
PEA     SRC2 address    ; push address of rate first
PEA     SRC  address    ; push address of number of periods second
PEA     DST  address    ; push address of destination third
<EOPMACRO>
```

to effect

DST <-- <op> (SRC2, SRC)

where <op> is compound or annuity, SRC2 is the rate, and SRC is the number of periods. All arguments SRC2, SRC, and DST must be of the extended type.

Example

C <-- $(1 + R)^N$, where C, R, and N are of type extended.

```
PEA     R_ADR   ; push address of R
PEA     N_ADR   ; push address of N
PEA     C_ADR   ; push address of C
FCOMPOUND       ; compound
```

Appendix A
68000 SANE Access

In your assemblies include the file TLASM/SANEMACS.TEXT, which contains the macros mentioned in this manual. The standard version is for Macintosh. For programs that will run on Lisa, redefine the symbol FPBYTRAP as follows:

```
FPBYTRAP .EQU   0
```

On Macintosh, the object code for FP68K and ELEMS68K is automatically loaded as needed by the Package Manager. On Lisa, it suffices to link your assembled code with the intrinsic unit file IOSFPLIB.OBJ.

Appendix B
68000 SANE Macros

```
;---------------------------------------------------------------
;
; FILE: SANEMACS.TEXT
;
;  These macros and equates give assembly language access to
;  the 68K floating-point arithmetic routines.
;---------------------------------------------------------------

;---------------------------------------------------------------
; WARNING: set FPBYTRAP for your system.
;---------------------------------------------------------------
FPBYTRAP        .EQU    1       ;0 for Lisa, 1 for Macintosh

        .MACRO  JSRFP
            .IF     FPBYTRAP
                _FP68K          ;defined in TOOLMACS
            .ELSE
                .REF    FP68K
                JSR     FP68K
            .ENDC
        .ENDM

        .MACRO  JSRELEMS
            .IF     FPBYTRAP
                _ELEMS68K       ;defined in TOOLMACS
            .ELSE
                .REF    ELEMS68K
                JSR     ELEMS68K
            .ENDC
        .ENDM

;---------------------------------------------------------------
; Operation code masks.
;---------------------------------------------------------------
FOADD           .EQU    $0000   ; add
FOSUB           .EQU    $0002   ; subtract
FOMUL           .EQU    $0004   ; multiply
FODIV           .EQU    $0006   ; divide
FOCMP           .EQU    $0008   ; compare, no exception from unordered
FOCPX           .EQU    $000A   ; compare, signal invalid if unordered
```

```
FOREM           .EQU    $000C   ; remainder
FOZ2X           .EQU    $000E   ; convert to extended
FOX2Z           .EQU    $0010   ; convert from extended
FOSQRT          .EQU    $0012   ; square root
FORTI           .EQU    $0014   ; round to integral value
FOTTI           .EQU    $0016   ; truncate to integral value
FOSCALB         .EQU    $0018   ; binary scale
FOLOGB          .EQU    $001A   ; binary log
FOCLASS         .EQU    $001C   ; classify
; UNDEFINED     .EQU    $001E

FOSETENV        .EQU    $0001   ; set environment
FOGETENV        .EQU    $0003   ; get environment
FOSETHV         .EQU    $0005   ; set halt vector
FOGETHV         .EQU    $0007   ; get halt vector
FOD2B           .EQU    $0009   ; convert decimal to binary
FOB2D           .EQU    $000B   ; convert binary to decimal
FONEG           .EQU    $000D   ; negate
FOABS           .EQU    $000F   ; absolute
FOCPYSGNX       .EQU    $0011   ; copy sign
FONEXT          .EQU    $0013   ; next-after
FOSETXCP        .EQU    $0015   ; set exception
FOPROCENTRY     .EQU    $0017   ; procedure entry
FOPROCEXIT      .EQU    $0019   ; procedure exit
FOTESTXCP       .EQU    $001B   ; test exception
; UNDEFINED     .EQU    $001D
; UNDEFINED     .EQU    $001F

;-------------------------------------------------------------
; Operand format masks.
;-------------------------------------------------------------
FFEXT           .EQU    $0000   ; extended -- 80-bit float
FFDBL           .EQU    $0800   ; double   -- 64-bit float
FFSGL           .EQU    $1000   ; single   -- 32-bit float
FFINT           .EQU    $2000   ; integer  -- 16-bit integer
FFLNG           .EQU    $2800   ; long int -- 32-bit integer
FFCOMP          .EQU    $3000   ; comp     -- 64-bit integer

;-------------------------------------------------------------
; Precision code masks: forces a floating point output
; value to be coerced to the range and precision specified.
;-------------------------------------------------------------
FCEXT           .EQU    $0000   ; extended
FCDBL           .EQU    $4000   ; double
FCSGL           .EQU    $8000   ; single
```

```
;-----------------------------------------------------------
; Operation macros: operand addresses should already be on
; the stack, with the destination address on top.  The
; suffix X, D, S, C, I, or L  determines the format of the
; source operand -- extended, double, single, comp,
; integer, or long integer, respectively; the destination
; operand is always extended.
;-----------------------------------------------------------


;-----------------------------------------------------------
; Addition.
;-----------------------------------------------------------
        .MACRO  FADDX
        MOVE.W  #FFEXT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDD
        MOVE.W  #FFDBL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDS
        MOVE.W  #FFSGL+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDC
        MOVE.W  #FFCOMP+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDI
        MOVE.W  #FFINT+FOADD,-(SP)
        JSRFP
        .ENDM

        .MACRO  FADDL
        MOVE.W  #FFLNG+FOADD,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------
; Subtraction.
;-----------------------------------------------------------
        .MACRO  FSUBX
```

```
        MOVE.W  #FFEXT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBD
        MOVE.W  #FFDBL+FOSUB,-(SP)
        JSRFP
        .ENDM
        .MACRO  FSUBS
        MOVE.W  #FFSGL+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBC
        MOVE.W  #FFCOMP+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBI
        MOVE.W  #FFINT+FOSUB,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSUBL
        MOVE.W  #FFLNG+FOSUB,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Multiplication.
;-------------------------------------------------------------
        .MACRO  FMULX
        MOVE.W  #FFEXT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULD
        MOVE.W  #FFDBL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULS
        MOVE.W  #FFSGL+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULC
```

```
        MOVE.W  #FFCOMP+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULI
        MOVE.W  #FFINT+FOMUL,-(SP)
        JSRFP
        .ENDM

        .MACRO  FMULL
        MOVE.W  #FFLNG+FOMUL,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Division.
;-------------------------------------------------------------
        .MACRO  FDIVX
        MOVE.W  #FFEXT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVD
        MOVE.W  #FFDBL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVS
        MOVE.W  #FFSGL+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVC
        MOVE.W  #FFCOMP+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVI
        MOVE.W  #FFINT+FODIV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDIVL
        MOVE.W  #FFLNG+FODIV,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------------
; Square root.
;-----------------------------------------------------------------
        .MACRO  FSQRTX
        MOVE.W  #FOSQRT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Round to integer, according to the current rounding mode.
;-----------------------------------------------------------------
        .MACRO  FRINTX
        MOVE.W  #FORTI,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Truncate to integer, using round toward zero.
;-----------------------------------------------------------------
        .MACRO  FTINTX
        MOVE.W  #FOTTI,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Remainder.
;-----------------------------------------------------------------
        .MACRO  FREMX
        MOVE.W  #FFEXT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMD
        MOVE.W  #FFDBL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMS
        MOVE.W  #FFSGL+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREMC
        MOVE.W  #FFCOMP+FOREM,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FREMI
        MOVE.W  #FFINT+FOREM,-(SP)
        JSRFP
        .ENDM

        .MACRO  FREML
        MOVE.W  #FFLNG+FOREM,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------------
; Logb.
;-------------------------------------------------------------------
        .MACRO  FLOGBX
        MOVE.W  #FOLOGB,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------------
; Scalb.
;-------------------------------------------------------------------
        .MACRO  FSCALBX
        MOVE.W  #FFINT+FOSCALB,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------------
; Copy-sign.
;-------------------------------------------------------------------
        .MACRO  FCPYSGNX
        MOVE.W  #FOCPYSGN,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------------
; Negate.
;-------------------------------------------------------------------
        .MACRO  FNEGX
        MOVE.W  #FONEG,-(SP)
        JSRFP
        .ENDM
```

```
;-----------------------------------------------------------------
; Absolute value.
;-----------------------------------------------------------------
        .MACRO  FABSX
        MOVE.W  #FOABS,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Next-after.  NOTE: both operands are of the same
; format, as specified by the usual suffix.
;-----------------------------------------------------------------
        .MACRO  FNEXTS
        MOVE.W  #FFSGL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTD
        MOVE.W  #FFDBL+FONEXT,-(SP)
        JSRFP
        .ENDM

        .MACRO  FNEXTX
        MOVE.W  #FFEXT+FONEXT,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Conversion to extended.
;-----------------------------------------------------------------
        .MACRO  FX2X
        MOVE.W  #FFEXT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2X
        MOVE.W  #FFDBL+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2X
        MOVE.W  #FFSGL+FOZ2X,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FI2X
        MOVE.W  #FFINT+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2X
        MOVE.W  #FFLNG+FOZ2X,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2X
        MOVE.W  #FFCOMP+FOZ2X,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Conversion from extended.
;-------------------------------------------------------------
        .MACRO  FX2D
        MOVE.W  #FFDBL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2S
        MOVE.W  #FFSGL+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2I
        MOVE.W  #FFINT+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2L
        MOVE.W  #FFLNG+FOX2Z,-(SP)
        JSRFP
        .ENDM

        .MACRO  FX2C
        MOVE.W  #FFCOMP+FOX2Z,-(SP)
        JSRFP
        .ENDM
```

```
;-------------------------------------------------------------
; Binary to decimal conversion.
;-------------------------------------------------------------
        .MACRO  FX2DEC
        MOVE.W  #FFEXT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FD2DEC
        MOVE.W  #FFDBL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FS2DEC
        MOVE.W  #FFSGL+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FC2DEC
        MOVE.W  #FFCOMP+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FI2DEC
        MOVE.W  #FFINT+FOB2D,-(SP)
        JSRFP
        .ENDM

        .MACRO  FL2DEC
        MOVE.W  #FFLNG+FOB2D,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Decimal to binary conversion.
;-------------------------------------------------------------
        .MACRO  FDEC2X
        MOVE.W  #FFEXT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2D
        MOVE.W  #FFDBL+FOD2B,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FDEC2S
        MOVE.W  #FFSGL+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2C
        MOVE.W  #FFCOMP+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2I
        MOVE.W  #FFINT+FOD2B,-(SP)
        JSRFP
        .ENDM

        .MACRO  FDEC2L
        MOVE.W  #FFLNG+FOD2B,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Compare, not signaling invalid on unordered.
;-------------------------------------------------------------
        .MACRO  FCMPX
        MOVE.W  #FFEXT+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPD
        MOVE.W  #FFDBL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPS
        MOVE.W  #FFSGL+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPC
        MOVE.W  #FFCOMP+FOCMP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCMPI
        MOVE.W  #FFINT+FOCMP,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FCMPL
        MOVE.W  #FFLNG+FOCMP,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Compare, signaling invalid on unordered.
;-----------------------------------------------------------------
        .MACRO  FCPXX
        MOVE.W  #FFEXT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXD
        MOVE.W  #FFDBL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXS
        MOVE.W  #FFSGL+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXC
        MOVE.W  #FFCOMP+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXI
        MOVE.W  #FFINT+FOCPX,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCPXL
        MOVE.W  #FFLNG+FOCPX,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; The following macros define a set of so-called floating
; branches.  They presume that the appropriate compare
; operation, macro FCMPz or FCPXz, precedes.
;-----------------------------------------------------------------
        .MACRO  FBEQ
        BEQ     %1
        .ENDM
```

```
.MACRO  FBLT
BCS     %1
.ENDM

.MACRO  FBLE
BLS     %1
.ENDM

.MACRO  FBGT
BGT     %1
.ENDM

.MACRO  FBGE
BGE     %1
.ENDM

.MACRO  FBULT
BLT     %1
.ENDM

.MACRO  FBULE
BLE     %1
.ENDM

.MACRO  FBUGT
BHI     %1
.ENDM

.MACRO  FBUGE
BCC     %1
.ENDM

.MACRO  FBU
BVS     %1
.ENDM

.MACRO  FBO
BVC     %1
.ENDM

.MACRO  FBNE
BNE     %1
.ENDM
```

```
        .MACRO  FBUE
        BEQ     %1
        BVS     %1
        .ENDM

        .MACRO  FBLG
        BNE     %1
        BVC     %1
        .ENDM

;-------------------------------------------------------------
; Short branch versions.
;-------------------------------------------------------------
        .MACRO  FBEQS
        BEQ.S   %1
        .ENDM

        .MACRO  FBLTS
        BCS.S     %1
        .ENDM

        .MACRO  FBLES
        BLS.S     %1
        .ENDM

        .MACRO  FBGTS
        BGT.S     %1
        .ENDM

        .MACRO  FBGES
        BGE.S     %1
        .ENDM

        .MACRO  FBULTS
        BLT.S     %1
        .ENDM

        .MACRO  FBULES
        BLE.S     %1
        .ENDM

        .MACRO  FBUGTS
        BHI.S     %1
        .ENDM
```

```
        .MACRO  FBUGES
        BCC.S     %1
        .ENDM

        .MACRO  FBUS
        BVS.S     %1
        .ENDM

        .MACRO  FBOS
        BVC.S     %1
        .ENDM

        .MACRO  FBNES
        BNE.S     %1
        .ENDM

        .MACRO  FBUES
        BEQ.S     %1
        BVS.S     %1
        .ENDM

        .MACRO  FBLGS
        BNE.S     %1
        BVC.S     %1
        .ENDM

;-----------------------------------------------------------------
; Class and sign inquiries.
;-----------------------------------------------------------------
FCSNAN          .EQU    1       ; signaling NAN
FCQNAN          .EQU    2       ; quiet NAN
FCINF           .EQU    3       ; infinity
FCZERO          .EQU    4       ; zero
FCNORM          .EQU    5       ; normal number
FCDENORM        .EQU    6       ; denormal number

        .MACRO  FCLASSS
        MOVE.W  #FFSGL+FOCLASS,-(SP)
        JSRFP
        .ENDM

        .MACRO  FCLASSD
        MOVE.W  #FFDBL+FOCLASS,-(SP)
        JSRFP
        .ENDM
```

```
        .MACRO  FCLASSX
        MOVE.W  #FFEXT+FOCLASS,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Bit indexes for bytes of floating point environment word.
;-----------------------------------------------------------------
FBINVALID       .EQU    0       ; invalid operation
FBUFLOW         .EQU    1       ; underflow
FBOFLOW         .EQU    2       ; overflow
FBDIVZER        .EQU    3       ; division by zero
FBINEXACT       .EQU    4       ; inexact
FBRNDLO         .EQU    5       ; low bit of rounding mode
FBRNDHI         .EQU    6       ; high bit of rounding mode
FBLSTRND        .EQU    7       ; last round result bit
FBDBL           .EQU    5       ; double precision control
FBSGL           .EQU    6       ; single precision control

;-----------------------------------------------------------------
; Get and set environment.
;-----------------------------------------------------------------
        .MACRO  FGETENV
        MOVE.W  #FOGETENV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETENV
        MOVE.W  #FOSETENV,-(SP)
        JSRFP
        .ENDM

;-----------------------------------------------------------------
; Test and set exception.
;-----------------------------------------------------------------
        .MACRO  FTESTXCP
        MOVE.W  #FOTESTXCP,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETXCP
        MOVE.W  #FOSETXCP,-(SP)
        JSRFP
        .ENDM
```

```
;-------------------------------------------------------------
; Procedure entry and exit.
;-------------------------------------------------------------
        .MACRO  FPROCENTRY
        MOVE.W  #FOPROCENTRY,-(SP)
        JSRFP
        .ENDM

        .MACRO  FPROCEXIT
        MOVE.W  #FOPROCEXIT,-(SP)
        JSRFP
        .ENDM

;-------------------------------------------------------------
; Get and set halt vector.
;-------------------------------------------------------------
        .MACRO  FGETHV
        MOVE.W  #FOGETHV,-(SP)
        JSRFP
        .ENDM

        .MACRO  FSETHV
        MOVE.W  #FOSETHV,-(SP)
        JSRFP
        .ENDM


;-------------------------------------------------------------
; Elementary function operation code masks.
;-------------------------------------------------------------
FOLNX           .EQU    $0000   ; base-e log
FOLOG2X         .EQU    $0002   ; base-2 log
FOLN1X          .EQU    $0004   ; ln (1 + x)
FOLOG21X        .EQU    $0006   ; log2 (1 + x)

FOEXPX          .EQU    $0008   ; base-e exponential
FOEXP2X         .EQU    $000A   ; base-2 exponential
FOEXP1X         .EQU    $000C   ; exp (x) - 1
FOEXP21X        .EQU    $000E   ; exp2 (x) - 1

FOXPWRI         .EQU    $8010   ; integer exponentiation
FOXPWRY         .EQU    $8012   ; general exponentiation
FOCOMPOUNDX     .EQU    $C014   ; compound
FOANNUITYX      .EQU    $C016   ; annuity

FOSINX          .EQU    $0018   ; sine
FOCOSX          .EQU    $001A   ; cosine
```

```
FOTANX          .EQU    $001C   ; tangent
FOATANX         .EQU    $001E   ; arctangent
FORANDOMX       .EQU    $0020   ; random


;-------------------------------------------------------------
; Elementary function macros.
;-------------------------------------------------------------
        .MACRO  FLNX            ; base-e log
        MOVE.W  #FOLNX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG2X          ; base-2 log
        MOVE.W  #FOLOG2X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLN1X           ; ln (1 + x)
        MOVE.W  #FOLN1X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FLOG21X         ; log2 (1 + x)
        MOVE.W  #FOLOG21X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXPX           ; base-e exponential
        MOVE.W  #FOEXPX,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP2X          ; base-2 exponential
        MOVE.W  #FOEXP2X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP1X          ; exp (x) - 1
        MOVE.W  #FOEXP1X,-(SP)
        JSRELEMS
        .ENDM

        .MACRO  FEXP21X         ; exp2 (x) - 1
        MOVE.W  #FOEXP21X,-(SP)
        JSRELEMS
        .ENDM
```

```
.MACRO  FXPWRI          ; integer exponential
MOVE.W  #FOXPWRI,-(SP)
JSRELEMS
.ENDM

.MACRO  FXPWRY          ; general exponential
MOVE.W  #FOXPWRY,-(SP)
JSRELEMS
.ENDM

.MACRO  FCOMPOUNDX      ; compound
MOVE.W  #FOCOMPOUNDX,-(SP)
JSRELEMS
.ENDM

.MACRO  FANNUITYX       ; annuity
MOVE.W  #FOANNUITYX,-(SP)
JSRELEMS
.ENDM

.MACRO  FSINX           ; sine
MOVE.W  #FOSINX,-(SP)
JSRELEMS
.ENDM

.MACRO  FCOSX           ; cosine
MOVE.W  #FOCOSX,-(SP)
JSRELEMS
.ENDM

.MACRO  FTANX           ; tangent
MOVE.W  #FOTANX,-(SP)
JSRELEMS
.ENDM

.MACRO  FATANX          ; arctangent
MOVE.W  #FOATANX,-(SP)
JSRELEMS
.ENDM

.MACRO  FRANDOMX        ; random number generator
MOVE.W  #FORANDOMX,-(SP)
JSRELEMS
.ENDM
```

```
;-----------------------------------------------------------------
; NaN codes.
;-----------------------------------------------------------------
NANSQRT .EQU     1    ; Invalid square root such as sqrt(-1).
NANADD  .EQU     2    ; Invalid addition such as +INF - +INF.
NANDIV  .EQU     4    ; Invalid division such as 0/0.
NANMUL  .EQU     8    ; Invalid multiply such as 0 * INF.
NANREM  .EQU     9    ; Invalid remainder or mod such as x REM 0.
NANASCBIN .EQU  17    ; Attempt to convert invalid ASCII string.
NANCOMP .EQU    20    ; Result of converting comp NaN to floating.
NANZERO .EQU    21    ; Attempt to create a NaN with a zero code.
NANTRIG .EQU    33    ; Invalid argument to trig routine.
NANINVTRIG .EQU 34    ; Invalid argument to inverse trig routine.
NANLOG  .EQU    36    ; Invalid argument to log routine.
NANPOWER .EQU   37    ; Invalid argument to x^i or x^y routine.
NANFINAN .EQU   38    ; Invalid argument to financial function.
NANINIT .EQU   255    ; Uninitialized storage.
;-----------------------------------------------------------------
```

68000 SANE
Quick Reference Guide

This Guide contains diagrams of the SANE data formats and the 68K SANE operations and environment word.

C.1 Data Formats

Each of the diagrams below is followed by the rules for evaluating the number v.

In each field of each diagram, the leftmost bit is the msb and the rightmost is the lsb.

Format Diagram Symbols

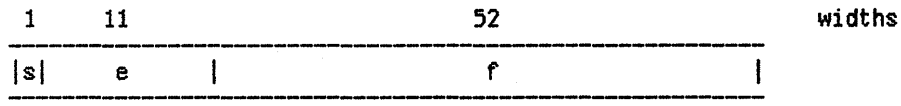
```
v       value of number
s       sign bit
e       biased exponent
i       explicit one's-bit (extended type only)
f       fraction
```

Single: 32 Bits

```
 1      8                          23                    widths
-------------------------------------------------------
|s|     e      |                    f                  |
-------------------------------------------------------
```

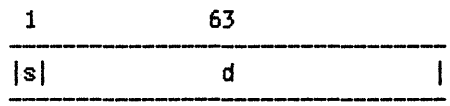
```
if 0 < e < 255,       then v = (-1)s * 2(e-127) * (1.f);
if e =   0 and f =/= 0, then v = (-1)s * 2(-126) * (0.f);
if e =   0 and f = 0, then v = (-1)s * 0;
if e = 255 and f = 0, then v = (-1)s * oo;
if e = 255 and f =/= 0, then v is a NaN.
```

Double: 64 Bits



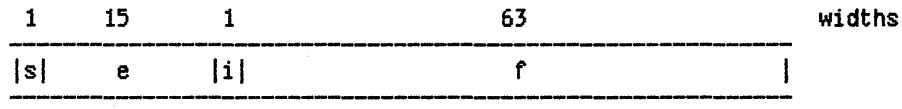

```
if 0 < e < 2047,        then v = (-1)s * 2(e-1023) * (1.f);
if e =    0 and f =+/- 0, then v = (-1)s * 2(-1022) * (0.f);
if e =    0 and f = 0,  then v = (-1)s * 0;
if e = 2047 and f = 0,  then v = (-1)s * oo;
if e = 2047 and f =+/- 0, then v is a NaN.
```

Comp: 64 Bits




```
if s = 1 and d = 0, then v is the unique comp NaN;
otherwise, v is the two's-complement value of the
64-bit representation.
```

Extended: 80 Bits




```
if 0 <= e < 32767,      then v = (-1)s * 2(e-16383) * (i.f);
if e = 32767 and f = 0, then v = (-1)s * oo, regardless of i;
if e = 32767 and f =+/- 0, then v is a NaN, regardless of i.
```

C.2 Operations

In the operations below, the operation's mnemonic is followed by the opword in parentheses: the first byte is the operation code; the second is the operand format code. For some operations, the first byte of the opword (xx) is ignored.

C.2.1 Abbreviations and Symbols

The symbols and abbreviations in this section closely parallel those in the text, although some are shortened. In some cases, the same symbol has various meanings, depending on context.

Operands

DST	destination operand (passed by address)
SRC	source operand (passed by address), pushed before DST
SRC2	second source operand (passed by address), pushed before SRC

Data Types

X	extended (80 bits)
D	double (64 bits)
S	single (32 bits)
I	integer (16 bits)
L	longint (32 bits)
C	comp (64 bits)
Dec	decimal Record
Decform	decform Record

68000 Processor Registers

D0	data register 0
X	extend bit of processor status register
N	negative bit of processor status register
Z	zero bit of processor status register
V	overflow bit of processor status register
C	carry bit of processor status register

Exceptions

I	invalid operation
U	underflow
O	overflow
D	divide-by-zero
X	inexact

For each operation, an exception marked with x indicates that the operation will signal the exception for some input.

Environment and Halts

```
EnWrd   SANE environment word (16-bit integer)
HltVctr SANE halt vector (32-bit longint)
```

C.2.2 Arithmetic Operations and Auxiliary Routines (Entry Point FP68K)

Operation	Operands and Data Types			Exceptions
ADD	DST	<-- DST +	SRC	I U O D X
FADDX (0000)	X	X	X	x - x - x
FADDD (0800)	X	X	D	x - x - x
FADDS (1000)	X	X	S	x - x - x
FADDC (3000)	X	X	C	x - x - x
FADDI (2000)	X	X	I	x - x - x
FADDL (2800)	X	X	L	x - x - x
SUBTRACT	DST	<-- DST -	SRC	I U O D X
FSUBX (0002)	X	X	X	x - x - x
FSUBD (0802)	X	X	D	x - x - x
FSUBS (1002)	X	X	S	x - x - x
FSUBC (3002)	X	X	C	x - x - x
FSUBI (2002)	X	X	I	x - x - x
FSUBL (2802)	X	X	L	x - x - x
MULTIPLY	DST	<-- DST *	SRC	I U O D X
FMULX (0004)	X	X	X	x x x - x
FMULD (0804)	X	X	D	x x x - x
FMULS (1004)	X	X	S	x x x - x
FMULC (3004)	X	X	C	x - x - x
FMULI (2004)	X	X	I	x - x - x
FMULL (2804)	X	X	L	x - x - x
DIVIDE	DST	<-- DST /	SRC	I U O D X
FDIVX (0006)	X	X	X	x x x x x
FDIVD (0806)	X	X	D	x x x x x
FDIVS (1006)	X	X	S	x x x x x
FDIVC (3006)	X	X	C	x x - x x
FDIVI (2006)	X	X	I	x x - x x
FDIVL (2806)	X	X	L	x x - x x

SQUARE ROOT	DST	<--	sqrt(DST)			I	U	O	D	X
FSQRTX (0012)	X		X			x	-	-	-	x
ROUND TO INT	DST	<--	rnd(DST)			I	U	O	D	X
FRINTX (0014)	X		X			x	-	-	-	x
TRUNC TO INT	DST	<--	chop(DST)			I	U	O	D	X
FTINTX (0016)	X		X			x	-	-	-	x
REMAINDER	DST	<--	DST REM	SRC		I	U	O	D	X
FREMX (000C)	X		X	X		x	-	-	-	-
FREMD (080C)	X		X	D		x	-	-	-	-
FREMS (100C)	X		X	S		x	-	-	-	-
FREMC (300C)	X		X	C		x	-	-	-	-
FREMI (200C)	X		X	I		x	-	-	-	-
FREML (280C)	X		X	L		x	-	-	-	-
	DO	<--	integer quotient DST/SRC, between -127 and +127							
LOG BINARY	DST	<--	logb(DST)			I	U	O	D	X
FLOGBX (001A)	X		X			x	-	-	x	-
SCALE BINARY	DST	<--	DST * 2^SRC			I	U	O	D	X
FSCALBX (0018)	X		X	I		x	x	x	-	x
NEGATE	DST	<--	-DST			I	U	O	D	X
FNEGX (000D)	X		X			-	-	-	-	-
ABSOLUTE VALUE	DST	<--	|DST|			I	U	O	D	X
FABSX (000F)	X		X			-	-	-	-	-
COPY-SIGN	SRC	<--	SRC with DST's sign			I	U	O	D	X
FCPYSGNX (0011)	XDorS		XDorS	XDorS		-	-	-	-	-
NEXT-AFTER	SRC	<--	next after	SRC	toward DST	I	U	O	D	X
FNEXTX (0013)	X			X	X	x	x	x	-	x
FNEXTD (0813)	D			D	D	x	x	x	-	x
FNEXTS (1013)	S			S	S	x	x	x	-	x

C.2.3 Conversions (Entry Point FP68K)

Operation	Operands and Data Types				Exceptions
CONVERT					
Bin to Bin	DST	<--	SRC		I U O D X
FX2X (0010)	X		X		x - - - -
FX2D (0810)	D		X		x x x - x
FX2S (1010)	S		X		x x x - x
FX2C (3010)	C		X		x - - - x
FX2I (2010)	I		X		x - - - x
FX2L (2810)	L		X		x - - - x
FD2X (080E)	X		D		x - - - -
FS2X (100E)	X		S		x - - - -
FC2X (300E)	X		C		- - - - -
FI2X (200E)	X		I		- - - - -
FL2X (280E)	X		L		- - - - -
Bin to Dec	DST	<--	SRC according to	SRC2	I U O D X
FX2DEC (000B)	Dec		X	Decform	x - - - x
FD2DEC (080B)	Dec		D	Decform	x - - - x
FS2DEC (100B)	Dec		S	Decform	x - - - x
FC2DEC (300B)	Dec		C	Decform	- - - - x
FI2DEC (200B)	Dec		I	Decform	- - - - x
FL2DEC (280B)	Dec		L	Decform	- - - - x
(First SRC2 is pushed, then SRC, then DST.)					
Dec to Bin	DST	<--	SRC		I U O D X
FDEC2X (0009)	X		Dec		- x x - x
FDEC2D (0809)	D		Dec		- x x - x
FDEC2S (1009)	S		Dec		- x x - x
FDEC2C (3009)	C		Dec		x - - - x
FDEC2I (2009)	I		Dec		x - - - x
FDEC2L (2809)	L		Dec		x - - - x

C.2.4 Compare and Classify (Entry Point FP68K)

Operation	Operands and Data Types	Exceptions

COMPARE

No invalid for unordered	Status Bits where DST	⟨-- ⟨relation⟩ ⟨relation⟩	SRC	I U O D X
FCMPX (0008)	X		X	x - - - -
FCMPD (0808)	X		D	x - - - -
FCMPS (1008)	X		S	x - - - -
FCMPC (3008)	X		C	x - - - -
FCMPI (2008)	X		I	x - - - -
FCMPL (2808)	X		L	x - - - -

(invalid only for signaling NaN inputs)

Signal invalid if unordered	Status Bits where DST	⟨-- ⟨relation⟩ ⟨relation⟩	SRC	I U O D X
FCPXX (000A)	X		X	x - - - -
FCPXD (080A)	X		D	x - - - -
FCPXS (100A)	X		S	x - - - -
FCPXC (300A)	X		C	x - - - -
FCPXI (200A)	X		I	x - - - -
FCPXL (280A)	X		L	x - - - -

⟨relation⟩	Status Bits X	N	Z	V	C
DST > SRC	0	0	0	0	0
DST < SRC	1	1	0	0	1
DST = SRC	0	0	1	0	0
DST & SRC unordered	0	0	0	1	0

CLASSIFY ⟨class⟩ ⟨-- class of SRC; ⟨sign⟩ ⟨-- sign of SRC; DST ⟨-- (-1)^⟨sign⟩ * ⟨class⟩ — I U O D X

FCLASSX (001C)	I	X	- - - - -
FCLASSD (081C)	I	D	- - - - -
FCLASSS (101C)	I	S	- - - - -

SRC	<class>	SRC	<sign>
signaling NaN	1	positive	0
quiet NaN	2	negative	1
infinite	3		
zero	4		
normalized	5		
denormalized	6		

C.2.5 Environmental Control (Entry Point FP68K)

Operation	Operands and Data Types	Exceptions
GET ENVIRONMENT	DST <-- EnvWrd	I U O D X
FGETENV (0003)	I	- - - - -
SET ENVIRONMENT	EnvWrd <-- SRC	I U O D X
FSETENV (0001)	I	x x x x x
(exceptions set by set-environment cannot cause halts)		
TEST EXCEPTION	Zbit <-- SRC Xcps clear	I U O D X
FTESTXCP (001B)	I	- - - - -
SET EXCEPTION	EnvWrd <-- EnvWrd AND SRC	I U O D X
FSETXCP (0015)	I	x x x x x
PROCEDURE ENTRY	DST <-- EnvWrd, EnvWrd <-- 0	I U O D X
FPROCENTRY (0017)	I	x x x x x
PROCEDURE EXIT	EnvWrd <-- SRC AND current Xcps	I U O D X
FPROCEXIT (0019)	I	x x x x x

C.2.6 Halt Control (Entry Point FP68K)

SET HALT VECTOR	HltVctr	<--	SRC	I U O D X
FSETHV (xx05)			L	- - - - -
GET HALT VECTOR	DST	<--	HltVctr	I U O D X
FGETHV (0007)	L			- - - - -

C.2.7 Elementary Functions (Entry Point ELEMS68K)

Operation	Operands and Data Types			Exceptions
BASE-E LOGARITHM	DST	<--	ln(DST)	I U O D X
FLNX (0000)	X		X	x - - x x
BASE-2 LOGARITHM	DST	<--	log2(DST)	I U O D X
FLOG2X (0002)	X		X	x - - x x
BASE-E LOG1 (LN1)	DST	<--	ln(1+DST)	I U O D X
FLN1X (0004)	X		X	x x - x x
BASE-2 LOG1	DST	<--	log2(1+DST)	I U O D X
FLOG21X (0006)	X		X	x x - x x
BASE-E EXPONENTIAL	DST	<--	e^DST	I U O D X
FEXPX (0008)	X		X	x x x - x
BASE-2 EXPONENTIAL	DST	<--	2^DST	I U O D X
FEXP2X (000A)	X		X	x x x - x
BASE-E EXP1	DST	<--	e^DST - 1	I U O D X
FEXP1X (000C)	X		X	x x x - x
BASE-2 EXP1	DST	<--	2^DST - 1	I U O D X
FEXP21X (000E)	X		X	x x x - x

INTEGER EXPONENTIATION	DST	<--	DST^SRC			I	U	O	D	X
FXPWRI (8010)	X		X	I		x	x	x	x	x
GENERAL EXPONENTIATION	DST	<--	DST^SRC			I	U	O	D	X
FXPWRY (8012)	X		X	X		x	x	x	x	x
COMPOUND INTEREST	DST	<--	compound(SRC2,SRC)			I	U	O	D	X
FCOMPOUND (C014)	X			X	X	x	x	x	x	x

(SRC2 is the rate; SRC is the number of periods.)

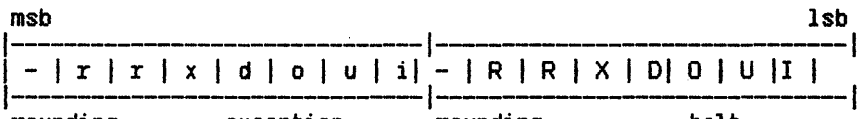
ANNUITY FACTOR	DST	<--	annuity(SRC2,SRC)			I	U	O	D	X
FANNUITY (C016)	X			X	X	x	x	x	x	x

(SRC2 is the rate; SRC is the number of periods.)

SINE	DST	<--	sin(DST)			I	U	O	D	X
FSINX (0018)	X		X			x	x	-	-	x
COSINE	DST	<--	cos(DST)			I	U	O	D	X
FCOSX (001A)	X		X			x	x	-	-	x
TANGENT	DST	<--	tan(DST)			I	U	O	D	X
FTANX (001C)	X		X			x	x	-	x	x
ARCTANGENT	DST	<--	atan(DST)			I	U	O	D	X
FATANX (001E)	X		X			x	x	-	-	x
RANDOM	DST	<--	random(DST)			I	U	O	D	X
FRANDX (0020)	X		X			x	x	x	-	x

C.3 Environment Word

The floating-point environment is encoded in the 16-bit integer format as shown below in hexadecimal:




```
rounding direction, bits 6000          rr
        0000 -- to-nearest
        2000 -- upward
        4000 -- downward
        6000 -- toward-zero

exception flags, bits 1F00
        0100 -- invalid                i
        0200 -- underflow              u
        0400 -- overflow               o
        0800 -- division-by-zero       d
        1000 -- inexact                x

rounding precision, bits 0060          RR
        0000 -- extended
        0020 -- double
        0040 -- single
        0060 -- UNDEFINED

halt enabled, bits 001F
        0001 -- invalid                I
        0002 -- underflow              U
        0004 -- overflow               O
        0008 -- division-by-zero       D
        0010 -- inexact                X
```

Bits 8000 and 0080 are undefined.

Note that the default environment is represented by the integer value zero.

The StdUnit

Contents

The StdUnit Unit

1 Introduction

StdUnit is the "Standard Unit," an intrinsic unit that provides a number of standard functions. It contains functions dealing with:

- Character and string manipulartion.
- File name manipulation.
- Prompting.
- Error messages.
- Special Workshop features.
- Conversions.

Workshop tools should use the unit wherever possible, especially for prompting and Operating System error reporting, to make the Workshop interface consistent.

Note: All names in StdUnit begin with the letters SU. This avoids name conflicts when incorporating the unit into your code and identifies where things come from.

2 Functional Areas

2.1 Initialization

StdUnit needs to be initialized before it can be used. Using the unit without initializing it will often result in an address or bus error.

2.2 String and Character Manipulation

StdUnit provides a standard string type, SUStr; a type for sets of characters; definitions for several standard characters (such as CR and BS); and procedures for case conversion, trimming blanks, and appending strings and characters.

2.3 File Name Manipulation

File name functions let you determine if a pathname is a volume or device name only; add extensions (such as .TEXT) to the file names (the procedure knows the conventions about when extensions should and should not be added); splitting a pathname into its three basic components--the device, volume, or catalog component, the file name component, and the extension component; putting the components back together into a file name; and modifying a file name given optional defaults for missing volume, file or extension components.

Note: Several of the procedures return overflow flags for identifying when a file name component has exceeded its character limit. You may choose to

ignore the overflow condition, particularly if you think it likely to occur only in perverse circumstances.

Note: The string parameters to these procedures are typed differently, sometimes SUStr's, or VAR SUStr's, or SUStrP's (pointers to SUStr's). This is to avoid problems with Pascal string typing when using the procedures with strings that are not SUStr's (e.g., PathName's), and to take into account the cases in which the parameters are likely to be string constants.

2.4 Prompting

StdUnit provides a number of procedures to get characters, strings, file names, integers, yes/no responses, etc., from the console, providing for default values where appropriate.

Most of the prompting procedures return a PrompState indicating whether an escape [CLEAR] was typed, whether the default was taken, or whether there was a request for options with ?. The states returned are given for each procedure. You can ignore the prompt states you are not interested in. For example, if you don't want to treat ? as an option request, you can ignore the SUOptions state and not treat the ? returned as a special character.

2.5 Error Text Retrieval

StdUnit provides a mechanism to retrieve single-line error messages from specially formatted error files. Error messages can be looked up by number in one or more error files.

You can use the OS error file OSErrs.ERR to return a real message when an OS error occurs (see Example 2, below). Note that OS errors are also returned via Pascal's IORESULT.

The ErrTool program lets you make your own compacted message files. Using this error mechanism, you can add and modify messages without recompiling your program. ErrTool is described in the *Workshop User's Guide*, Chapter 11, The Utilities.

A call to retrieve a message opens the error file, searches the directory for the error number, finds location of the message, and returns the text.

A program can use StdUnit to access more than one error file simultaneously. For example, your program can access different files for OS error messages and your own messages.

2.6 Workshop Support

Special Workshop functions let you:

- Stop the execution of an EXEC file in progress.
- Find out the name of the boot and current prefix volumes (SysVols).
- Use a super-RESET that will try to open a file first on the prefix volume, then on the boot volume, then on the current process volume.

2.7 Conversions

Conversion procedures let you convert from integers and longints to strings, and from strings to integers and longints.

3 Examples

Example 1

Assume we are going to prompt for an output file name (OutFName) and that we already have the input file name (InFName). We will use SUSplitFN to split the input file name into its various components. Then we will prompt for the output file name (with SUGetFN) using the volume and file name components of the input file name as defaults but with a .ERR extension. We then do a CASE on the prompt state (PState) returned by SUGetFN. The will terminate if the file specification is an escape [CLEAR]; say that no option are available if ? is typed as an option request; prompt again if no file is specified, since we want to require an output file; and fall through if the default is accepted or some other file is specified. Note that we only have to check for the prompt states we are interested in for special handling.

```
9999:
      WRITE ('Name of Error Output File ');
      SUSplitFN (@InFName, @VolN, @FN, @Ext);
      SUGetFN (@OutFName, PState, VolN, FN, '.ERR');
      CASE PState OF
        SUEscape:  EXIT (ErrFileP);  {exit from program}
        SUOptions: BEGIN
                     WRITELN ('No options are available. ');
                     GOTO 9999;
                   END;
        SUNone:    GOTO 9999;
      END;  {CASE}
```

Example 2

Suppose we have just made a Pascal I/O call and want to report an error (along with the OS message text) if we receive a nonzero IORESULT. Note that we copy IORESULT into our IOStatus variable so that the subsequent WRITELN will not reset the value of IORESUILT before we get a chance to use it. (EMsg should be a SUStr.)

```
IF IORESULT <> 0 THEN
  BEGIN
    IOStatus := IORESULT;
    WRITELN ('Error opening input file.');
    SUErrText ('OsErrs.ERR', IOStatus, @EMsg);
    WRITELN (EMsg);
  END;
```

4 Interface

```
{------------------------------ SU:StdUnit ------------------------------}
{ Copyright 1983, 1984, Apple Computer, Inc.                             }
{                                                                        }
{ This unit provides a number of standard type definitions and a collection }
{ of procedures which perform a variety of common functions.  The areas  }
{ covered are:                                                           }
{   (1) String and Character manipulation                                }
{   (2) File Name Manipulation                                           }
{   (3) Prompting                                                        }
{   (4) Retrieval of messages from disk                                  }
{   (5) Development System Support                                       }
{   (6) Conversions                                                      }
{                                                                        }
{ Fred Forsman 4-25-84                                                   }
{------------------------------------------------------------------------}

{$SETC ForOS11orHigher := TRUE}

{$R-} { make it fast, no range checking }
{$S SULib }

UNIT StdUnit;
  INTRINSIC;

  INTERFACE

    USES
      {$U libOS/SysCall.obj    } SysCall, { for definition of PathName, etc. }
      {$U libPL/PasLibCall.obj } PasLibCall,
      {$U libPL/PPasLibC.obj   } PPasLibC;

    CONST
      SUMaxStrLeng = 255;
      SUNullStr    = '';
      SUSpace      = ' ';
      SUOrdCR      = 13;
      SUMaxPNLeng  = 66;    { max length of path name }
      SUMaxVNLeng  = 33;    { max length of volume name, includes leading '-' }
      SUMaxFNLeng  = 32;    { maximum length of file name }
      SUVolSuffix  = '-';   { suffix or end of device or volume name }

    TYPE
      SUSetOfChar  = SET OF CHAR;
      SUStrP       = ^SUStr;
```

```
    SUStrP       = ^SUStr;
    SUStr        = STRING[255];
    SUVolName    = STRING[SUMaxVNLeng];
    SUFile       = FILE;
    SUFileP      = ^SUFile;
    PromptState = (SUDefault, { the default (if any) was chosen) }
                   SUEscape,  { the "Clear" key was pressed }
                   SUNone,    { nothing specified in response to prompt }
                   SUOptions, { "?" was entered--ie, an option query }
                   SUValid,   { valid reponse }
                   SUInvalid  { invalid reponse--eg, non-number to SUGetInt}
                   );
    ErrTextRet  = (SUOk,           { successful }
                   SUBadEFOpen,    { could not open error file }
                   SUBadEFRead,    { error reading error file }
                   SUErrNNotFound { error number not found }
                   );
    ConvNState  = (SUValidN,       { valid number }
                   SUNoN,          { no number -- nothing specified }
                   SUBadN,         { invalid number }
                   SUNOverFlow     { overflow -- number too big }
                   );

  VAR
    SUOsBootV  : SUVolName;    { The volume the OS was booted from }
    SUMyProcV  : SUVolName;    { The volume MyProcess was started from }
    SUBell, SUBackSpace, SUCr, SUTab, SUEsc,
      SUDle, SUNul : CHAR;     { predefined ch vars } {ff 1/23/84}
    SUNullS    : SUStr;        { predefined str var }
    SUKeyBoard : INTERACTIVE; { non-echoing console, used by SUGetCh }
                                                        {ff 2/29/84}

{============================== INIT AND DONE ==============================}

  PROCEDURE SUInit;
    { Should be called before using rest of unit.  On the OS this opens
      "-KeyBoard".  It also initializes the standard character variables. }

  PROCEDURE SUDone;
    { Can be called when done using unit (although this is not strictly
      necessary.  On the OS this closes "-KeyBoard". }

{============================ STRINGS AND CHARS ============================}

  FUNCTION SUUpCh (Ch : CHAR) : CHAR;
    { SUUpCh returns the ch that was passed, uppercased if it was lower
      case. }
```

```
FUNCTION SULowCh (Ch : CHAR) : CHAR;
  { SULowCh returns the ch that was passed, lowercased if it was upper
    case. }

PROCEDURE SUUpStr (S: SUStrP);
  { SUUpStr uppercases the string that is passed. }

PROCEDURE SULowStr (S: SUStrP);
  { SULowStr lowercases the string that is passed. }

FUNCTION SUEqStr (S1: SUStrP; S2: SUStrP) : BOOLEAN;          {ff 2/29/84}
  { SUEqStr returns TRUE if the two strings are equal (ignoring case). }

FUNCTION SUEq2Str (S1: SUStrP; S2: SUStr) : BOOLEAN;          {ff 3/7/84}
  { SUEq2Str returns TRUE if the two strings are equal (ignoring case).
    This variant of SUEqStr allows the second parameter to be a constant.}

PROCEDURE SUTrimLeading (S: SUStrP);                          {ff 2/29/84}
  { SUTrimLeading removes the leading blanks and tabs in the passed
    string. }

PROCEDURE SUTrimTrailing (S: SUStrP);                         {ff 2/29/84}
  { SUTrimTrailing removes the trailing blanks and tabs in the passed
    string. }

PROCEDURE SUTrimBlanks (S: SUStrP);
  { SUTrimBlanks removes leading and trailing blanks and tabs in the
    passed string. }

PROCEDURE SUAddCh (S: SUStrP; Ch : CHAR; MaxStrLeng : INTEGER;
                   VAR OverFlow : BOOLEAN);
  { SUAddCh appends the passed ch to the end of the passed string.
    OverFlow is set to TRUE if adding the ch will cause the string to be
    longer than MaxStrLeng. }

PROCEDURE SUConcat (S1: SUStrP; S2: SUStrP);
  { SUConcat appends the second passed str to the end of the first passed
    string.  It is assumed that the target string is of sufficient size to
    accomodate the new value. }

PROCEDURE SUAddStr (S1: SUStrP; S2: SUStrP; MaxStrLeng : INTEGER;
                    VAR OverFlow : BOOLEAN);
  { SUAddStr appends the second passed str to the end of the first passed
    string.  OverFlow is set to TRUE if adding the second string will
    cause the resulting string to be longer than MaxStrLeng. }
```

```
    PROCEDURE SUSetStr (Dest: SUStrP; Src: SUStrP);
      { SUSetStr sets the target string (Dest) to the given value (Src) by
        copying the value onto the target.  It is assumed that the target
        string is of sufficient size to accomodate the new value. }

    PROCEDURE SUCopyStr (Dest: SUStrP; Src: SUStrP; Start, Count: INTEGER);
      { SUCopyStr sets the destination string (Dest) to the specified
        substring of the source string (Src) by copying the appropriate part
        of the source to the destination.  It is assumed that the destination
        string is of sufficient size to accomodate the new value, and that the
        Start and Count values are reasonable. }

{================================ FILE NAMES ================================}

    FUNCTION SUIsVolName (FN: SUStrP): BOOLEAN;
      { SUIsVolName returns a boolean indicating whether the passed file name,
        FN, is a volume or device name (i.e., not a full file name) }

    PROCEDURE SUVolPart (PathN: SUStrP; VolN: SUStrP);              {ff 2/29/84}
      { SUVolPart extracts the volume name part of a pathname (or catalog
        specification). }

    PROCEDURE SUAddExtension (FN: SUStrP; DefExt: SUStr;
                              MaxStrLeng: INTEGER; VAR OverFlow: BOOLEAN);
      { SUAddExtension will add the default extension, DefExt, to the end of
        the file name, S, if the extension is not already present.  If the
        file name ends with a dot, the dot will be removed and no extension
        will be added.  If the pathname is a device or volume name only no
        extension will be added.  OverFlow is set true if adding the extension
        will overflow the string (determined using MaxStrLeng). }

    PROCEDURE SUSplitFN (PathN: SUStrP; CatN: SUStrP; FN: SUStrP;
                         Ext: SUStrP);
      { SUSplitFN splits a PathName into its catalog, file name, and file
        name extension components. }

    PROCEDURE SUMakeFN (PathN: SUStrP; CatN: SUStrP; FN: SUStrP; Ext: SUStr;
                        VAR OverFlow: BOOLEAN);
      { SUMakeFN constructs a PathName from its catalog, file name, and
        file name extension components.  The OS CatN's are assumed to have a
        leading "-".  OverFlow is set if any of the file name components are
        too long.  This procedure will not create a file name over SUMaxPNLeng
        chars long.}

    PROCEDURE SUChkFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
                       DefFN: SUStr; DefExt: SUStr);
      { SUChkFN checks a file name specification, putting result type in
```

```
        PState.  If no file name is given, then DefFN is used.  If FN does not
        have DefExt  in it, then the extension is appended.  If no volume is
        specifed then  the DefVol is used.  PState is set appropriately:
          PState = SUOptions  if '?' is hit to ask for options
          PState = SUDefault  if nothing specified when a default is present
          PState = SUNone     if default overriden with '\' or if CR with no
                              default
          PState = SUInvalid  if one or more of the file name components
                              overflowed
          PState = SUValid    otherwise }

{================================ PROMPTING ================================}

    PROCEDURE SUGetCh (VAR Ch: CHAR);
      { SUGetCh reads a character from the console without echoing it and }
      { without interpreting <cr> as <sp>, as Read (Ch) does. }

    PROCEDURE SUGetLine (S: SUStrP; VAR PState: PromptState);
      { SUGetLine reads a line from the console a character at a time,
        performing its own line editing.  PState is set appropriately:
          PState = SUEscape  if <clear> was hit.
          PState = SUValid   otherwise. }

    PROCEDURE SUGetStr (S: SUStrP; VAR PState: PromptState; DefVal: SUStr);
      { SUGetStr reads a string from the console; it is like SUGetLine with
        the addition of defaults.  PState is set appropriately:
          PState = SUDefault if <cr> only was hit; S is set to DefVal.
          PState = SUEscape  if <clear> was the first character hit.
          PState = SUValid   otherwise. }

    PROCEDURE SUGetFN (FN: SUStrP; VAR PState: PromptState; DefVol: SUStr;
      DefFN: SUStr; DefExt: SUStr);
        { SUGetFN reads a file name from the console, with result type in
          PState.  SUGetFN will print out any defaults in brackets (such as
          [FOO] [.TEXT]) before prompting for the file name.  If no file name
          is given, then DefFN is used.  If FN does not have DefExt in it,
          then the extension is appended.  If no volume is specifed then the
          DefVol is used.  PState is set appropriately:
            PState = SUEscape   if <clear> hit
            PState = SUOptions  if '?' is hit to ask for options
            PState = SUDefault  if nothing specified when a default is present
            PState = SUNone     if default overriden with '\' or if CR with no
                                default
            PState = SUInvalid  if one or more of the file name components
                                overflowed
            PState = SUValid    otherwise }
```

```
    PROCEDURE SUGetInt (VAR I: INTEGER; VAR PState: PromptState;
                        DefVal: INTEGER);
      { SUGetInt reads an INTEGER from the console, with PState set as in
        SUGetStr, except that PState = SUInvalid when a non-numeric is input.}

    PROCEDURE SUWaitEscOrSp (VAR PState: PromptState);
      { SUWaitEscOrSp prints a message 'Type <space> to continue, <clear> to
        exit.' & waits for the user to hit a <sp> or <clear>, setting PState
        appropriately:
          PState = SUEscape if <clear> was hit
          PState = SUValid  if <sp> was hit }

    PROCEDURE SUWaitSp;
      { SUWaitSp prints a message ('Type <space> to continue.') and waits for
        the user to hit a <sp>. }

    PROCEDURE SUGetChInSet (VAR Ch: CHAR; Chars: SUSetOfChar);
      { SUGetChInSet reads characters from the console (without echoing) until
        a character from the given set is typed.  The accepted character is
        echoed and an end-of-line is written.  The character matching ignores
        case. }

    FUNCTION SUGetYesNo : BOOLEAN;
      { SUGetYesNo prints the message "(Y or N)" and reads characters from the
        console (without echoing) until a 'y', 'Y', 'n', or 'N' is typed.  If
        a 'y' is typed "Yes" will be printed followed by an end-of-line; if
        'n' is typed "No" will be printed.  The appropriate boolean value is
        returned. }

    FUNCTION SUGetBool (Default: BOOLEAN): BOOLEAN;
      { SUGetBool prints the message "(Y or N)  [<default>]" and reads
        characters from the  console (without echoing) until a 'y', 'Y', 'n',
        'N', space or return is typed.  If a 'y' is typed "Yes" will be
        printed in the place of the default.  If 'n' is typed "No" will be
        printed.  If a space or return is typed the default is used.  The
        appropriate boolean value is returned. }

{========================== ERROR TEXT RETRIEVAL ==========================}

    PROCEDURE SUGetErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP;
                            VAR ErrRet: ErrTextRet);
      { SUGetErrText retrieves error message text, given an error number and
        and error file to look the error up in.  The error file should have
        been generated by the error file processor.  SUGetErrText use
        SUSysReset to open the error file. }

    PROCEDURE SUErrText (ErrFN: SUStr; ErrN: INTEGER; ErrMsg: SUStrP);
```

```
    { SUErrText retrieves error message text, just as does SUGetErrText;
      however, if the text is not obtainable due to a non-SUOk ErrRet value
      from SUErrText, SUErrText will return the string
        "Error message text not available." }

{============================== DEV. SYS. SUPPORT ==============================}

  PROCEDURE SUStopExec (VAR ErrNum: INTEGER);
    { Should be called to stop the current exec file if an error occurs in a
      program running under an exec.  Returns any error conditions
      encountered in closing the exec file in the errnum var parameter.
      Informs the shell that the exec file was terminated due to an error. }

  PROCEDURE SUCloseExec (VAR ErrNum: INTEGER);  {ff 3/7/84}
    { Should be called to stop the current exec file only if you want to do
      so without informing the shell that the exec file was terminated due
      to an error.  You should probably use SUStopExec unless you have a
      good reason to use this alternate version. }

  PROCEDURE SUInitSysVols;
    { Initializes "SUMyProcV" and "SUOsBootV", the name of the volume on
      which my process was created and the name of the volume which the OS
      was booted off of.  A message may be printed if there is trouble
      getting this information from the OS.  This can be called more than
      once; it will only make the OS calls the first time. }

  PROCEDURE SUSysReset (F : SUFileP; FN : SUStr; VAR IOStatus : INTEGER);
    { SUSysReset is for opening system files, and will try the prefix, boot,
      and current process volumes (in that order) when trying to access a
      file.  SUSysReset assumes that the file name FN does not have a volume
      name.  SUSysReset may sometimes have to call SUInitSysVols. }

{============================== CONVERSIONS ================================}

  PROCEDURE SUIntToStr (N : INTEGER; S : SUStrP);
    { SUIntToStr converts an integer into its string form;  The string which
      S points to should be of length >= 6 (5 digits + sign). }

  PROCEDURE SULIntToStr (N : LONGINT; S : SUStrP);
    { SULIntToStr converts an longint into its string form;  The string
      which S points to should be of length >= 11 (10 digits + sign). }

  PROCEDURE SUStrToInt (NS : SUStrP; VAR N : INTEGER;
                        VAR CState : ConvNState);
    { SUStrToInt converts a string to an INTEGER.  Leading and trailing
      blanks and tabs are permitted.  A leading sign ['-', '+'] is
      permitted.  The CState variable (conversion state) will be set to
```

```
    indicate if the number was valid, if no number was present, if an
    invalid number was specified, or if the number overflowed. }

PROCEDURE SUStrToLInt (NS : SUStrP; VAR N : LONGINT;
                       VAR CState : ConvNState);
  { SUStrToLInt converts a string to a LONGINT.  It behaves just like
    SUStrToInt otherwise. }
```

The ProgComm Unit

Contents

The ProgComm Unit

1 Introduction

ProgComm is an intrinsic unit in SULib that allows programs to communicate with the shell and with other programs. Three basic mechanisms are provided:

- *Set-Next-Run Command.* A program can tell the Workshop shell what to run next. The specified program will be run after the current program is done, taking precedence over even an exec file in progress.
- *The Program Return String.* The return string can be set by your program and accessed from the exec processor (via the RETSTR function). This allows exec scripts to be written that make choices based on program results.
- *The Communication Buffer.* The communication buffer is a 1K byte buffer global to the Workshop for communication between programs. A set of primitives supporting character- and line-oriented I/O to and from the buffer is provided.

These mechanisms can be used in conjunction with each other. For example, a program can write a series of invocation arguments to the communication buffer and then tell the shell which program to run next. This second program can check the communication buffer to find its arguments. Programs can be written so that, by convention, they first check the communication buffer for their arguments, and then prompt for input from the console only if the arguments are not found in the buffer.

2 ProgComm Routines

This section describes the ProgComm unit interface.

2.1 Initialization

The PCInit procedure initializes the ProgComm unit so that a program may use it.

Procedure PCInit;

PCInit should be called before using the ProgComm unit. The program's return string (RETSTR in the exec language) is initialized to the null string.

2.2 Set-Next-Run and the Return String

The PCSetRunCmd and PCSetRetStr procedures let a program set what program will run next and pass back a return string to the exec processor. The SUStr type comes from the Standard Unit (StdUnit in SULib), which provides a number of string-manipulation routines.

Procedure PCSetRunCmd (RC : SUStr);
PCSetRunCmd lets a program tell the shell what program or exec file to run after the current program terminates, allowing program chaining. RC, the run command passed to PCSetRunCmd, should be a string with the same program pathname or exec file invocation you would give to the Workshop Run command. The run command set in this way will take precedence over any keyboard type-ahead and over any pending exec file commands.

If you want to use PCSetRunCmd to run a Workshop tool normally invoked from the Workshop menu line, set RC to the two-character string consisting of an escape (CHR(27)) and the appropriate menu command letter. This is necessary because typing *E* to invoke the Editor is not always the same as saying Run Editor.OBJ. The Run command looks for Editor.OBJ on the three prefix volumes, while the E menu command looks on the Workshop boot volume first and then on the prefix volumes. (Note that only some items in the Workshop menu are actually separate tools that can be Run.)

Starting to run an exec file while you are already running another exec file causes the first one to be terminated so the second can run. This means that if exec file A runs program P, and P calls PCSetRunCmd to run exec file B, then, when program P terminates, exec file A will also be terminated so exec file B can run. Exec file A will not be resumed when exec file B has completed.

Procedure PCSetRetStr (RS : SUStr);
PCSetRetStr lets a program set a return string that can be accessed through the exec processor's RETSTR function. This lets exec files make choices based on information passed back to the shell by cooperating programs. How the return string is used and interpreted is up to you, and depends on what sort of information you want to pass back to the exec processor.

2.3 The Communication Buffer

The following procedures and functions operate on the *communication buffer*, a 1K byte buffer global to the Workshop shell (that is, it stays around between program invocations). The buffer can hold any type of information; a standard set of functions is provided for Pascallike character- or line-oriented access to the buffer.

Following are some constant, type, and variable declarations from the ProgComm interface which relate to the communication buffer.

```
CONST
  { communication buffer content types }
  PCNone    = -1;    { nothing in buffer }
  PCAny     = 0;     { for PCReset to match any content type  }
  PCText    = 1;     { text, as supported by PCGets & PCPuts  }
  PCBufrMax = 1023;  { max buffer index, ie, bufr is 1K bytes }
```

```
TYPE
  PCBufrP     = ^PCBufr; { pointer to bufr }
  PCBufr      = PACKED ARRAY [0..PCBufrMax] OF CHAR;
VAR
  PCBufrPtr   : PCBufrP; { points to bufr after successful open }
```

The communication buffer is given a *type* when it is opened for writing with PCReWrite. This type will be used to determine whether a potential reader trying to open the buffer with PCReset will be successful. The intent is to prevent reading of the buffer when the contents are not of the type expected by the reader. Three predefined constants are provided for buffer-typing: PCNone means that the buffer has no contents; PCText means that the buffer contains standard text with CR line delimiters; and PCAny matches any type, allowing a reader to override the typing mechanism. Other buffer content types (such as mouse events) may be defined by users, choosing a number to identify the new type that doesn't conflict with the predefined types. The only restriction is that communicating programs must have compatible conventions. To use the buffer for something other than text, use PCBufrPtr to access the buffer (using whatever means of interpretation of the buffer is desired).

The buffer also has an *access key*, which functions in much the same way as the content type (i.e., writers set it and readers must match it to gain access to the buffer). The intent of the access key is to prevent programs from reading the buffer when they are not the intended recipient. The access key should be established by agreement between communicating programs. If a buffer writer does not care about preventing unintended access to the buffer, the null string can be used for the access key. Note that the access key is case sensitive.

Following are the routines for opening and closing the communication buffer.

Procedure PCReWrite (WriteType: INTEGER; Key: SUStr);
PCReWrite opens the communication buffer for writing. The content type and access key are set. PCBufrPtr is set to point to start of the communication buffer. A PCReWrite will override any previous use of the buffer; that is, it will flush any previous buffer contents. WriteType should be an integer identifying the type of data you plan to write to the buffer. If you are planning to use the text-oriented primitives provided, WriteType should be PCText; otherwise, WriteType should be some integer established by agreement between the communicating programs. Key should be a string also established by agreement between the communicating programs. A useful form of key is one that identifies the intended recipient, so that contents left in the buffer are not read inadvertently by programs for which they were not intended.

Function PCReset (ReadType: INTEGER; Key: SUStr): BOOLEAN;
PCReset opens the buffer for reading. The boolean result will indicate whether the open was successful. The open will fail if ReadType does not match the type set by the last buffer writer or if Key does not match the key set by the last writer.

Function PCClose (KillBufr: BOOLEAN; Key: SUStr): BOOLEAN;
PCClose will close (or empty) the communication buffer. If KillBufr is true, the buffer will be emptied. In general, the buffer can be read more than once (by multiple readers) if desired. If a reader is finished with the buffer and knows that no one else should read the buffer, PCClose should be called with KillBufr set to true. The call to PCClose will fail if the access key does not match. PCClose may be used to flush buffers that were written by someone else, as long as you know the access key. PCClose may be called without calling PCReset or PCReWrite first.

2.4 Reading from and Writing to the Communication Buffer

The following functions provide a text-oriented buffer facility with Pascallike character- and line-oriented reads and writes.

Function PCPutCh (Ch: CHAR): BOOLEAN;
PCPutCh puts a character into the buffer. The boolean result indicates whether the operation was successful. It fails if the buffer is full or if the buffer was never opened successfully for writing. Note that PCPutCh(CR) is equivalent to PCPutLine('').

Function PCGetCh (VAR Ch: CHAR): BOOLEAN;
PCGetCh gets a character from the buffer. The boolean result indicates whether the operation was successful. It fails if the buffer is empty or if the buffer was never opened successfully for reading.

Function PCPutLine (L: SUStr): BOOLEAN;
PCPutLine puts a line into the buffer. A CR is put in the buffer following the string passed to PCPutLine. The boolean result indicates whether the operation was successful. It fails if the buffer is full or if the buffer was never opened successfully for writing.

Function PCGetLine (VAR L: SUStr): BOOLEAN;
PCGetLine gets a line from the buffer, where a line is the text from the current buffer pointer to the next CR or the end of file (whichever comes first). The boolean result indicates whether the operation was successful. It fails if the buffer is empty or if the buffer was never opened successfully for reading.

2.5 Internal Workshop Function

You will notice the following function in the ProgComm interface; it is used for special-purpose communication between the Workshop shell and various Workshop tools.

Function PCShellCmd (Cmd: INTEGER; P: SUStrP): BOOLEAN;
For internal use by Workshop tools only. Don't use this function.

3 Interface

```
INTERFACE

  USES
    {$U StdUnit    } StdUnit,
    {$U ShellComm } ShellComm;

  CONST
    { communication buffer content types for use with PCReset and PCReWrite }
    PCNone     = -1;       { nothing in buffer }
    PCAny      = 0;        { for PCReset to match any buffer content type }
    PCText     = 1;        { text, as supported by PCGet's and PCPut's below }

    PCBufrMax = 1023;      { max Bufr index, ie, comm bufr is 1K bytes }

    { command constants for PCShellCmd }
    PC_SetReallyStop    = 1;    { determines if SUStopExec really stops exec
                                  files }                                {ff 3/7/84}
    PC_GetReallyStop    = 2;
    PC_SetUnSavedEdits = 6960; { tells if unsaved edits are left in the
                                  editor     }                           {ff 3/12/84}
    PC_GetUnSavedEdits = 8751;

  TYPE
    PCBufrP    = ^PCBufr; { ptr to communication buffer }
    PCBufr     = PACKED ARRAY [0..PCBufrMax] OF CHAR;

  VAR
    PCBufrPtr : PCBufrP; { will point to PCBufr after successful PCReset or
                           PCReWrite }

  PROCEDURE PCInit;
    { PCInit should be called before using the ProgComm unit.  One effect of
note is that the program's return string (RetStr) is initialized to the null
string. }
  PROCEDURE PCSetRunCmd (RC : SUStr);
    { PCSetRunCmd enables a program to tell the shell what program (or exec
file) to run after the current program terminates, which allows program
"chaining".  The run command set in this way will take precedence over any
keyboard type-ahead and over any pending exec file commands. }
  PROCEDURE PCSetRetStr (RS : SUStr);
    { PCSetRetStr allows a program to set a return string which may be
accessed via the Exec Processor's RETSTR funciton.  This allows exec files to
make choices based on information passed back to the shell by cooperating
```

programs. How the return string should be used and interpreted is up to you, and will depend on what sort of information you want to pass back to the exec processor. (But in order to be a good citizen it is probably best to follow whatever system-wide conventions emerge and prevail.) }

{ The following procedures and function operate on the COMMUNICATION BUFFER, which is a 1K byte buffer which is global to the Workshop shell. The buffer can hold essentially any type of information, but a standard set of functions is provided for Pascal-like character or line-oriented access to the buffer.

The communication buffer is given a TYPE when it is opened for writing with PCReWrite. This type will be used to determine whether a potential reader trying to open the buffer with PCReset will be successful. The intent is to prevent reading of the buffer when the contents are not of the type expected by the reader. Three predefined constants are provided for buffer typing (PCNone which means the buffer has no contents; PCText which means that it has standard text with CR line delimiters; and PCAny which will match any type, allowing a reader to override the typing mechanism). Other buffer content types (such a mouse events) may be defined by users, choosing some number to identify the new type which does not conflict with the predefined types. We make no attempt here to provide a complete set of predefined types; the issue is simply one of having compatible conventions (agreement) between communicating programs. To use the buffer for something other than text, the variable PCBufrPtr may be used to access the buffer (using whatever means of interpretation is desired).

The buffer also has an ACCESS KEY, which functions in very much the same way as the content type (ie, writers set it and readers must match it to gain access to the buffer). The intent of the access key is to prevent programs from reading the buffer when they are not the intended recipient. The access key, again, is something that should be established by agreement between the communicating programs. If a buffer writer does not care about preventing unintended access to the buffer, the null string can be used for the access key. Note that the access key is case sensitive. }

```
PROCEDURE PCReWrite (WriteType : INTEGER; Key : SUStr);
  { PCReWrite opens the buffer for writing. The contents type and access
key are set. PCBufrPtr is set to point to the communication buffer. }
FUNCTION  PCReset (ReadType : INTEGER; Key : SUStr): BOOLEAN;
  { PCReset opens the buffer for reading. The boolean result will indicate
whether the open succeeded. The open will fail if contents type and access
key do not match the type and key set by the last buffer writer.}
FUNCTION  PCClose (KillBufr : BOOLEAN; Key : SUStr): BOOLEAN;     {ff 2/2/84}
  { PCClose will close the buffer. If KillBufr is true the buffer will be
emptied. In general, the buffer can be read more than once (by multiple
readers) if desired. If a reader is finished with the buffer and knows that
no one else should read the buffer, PCClose should be called with KillBufr set
to true. The call to PCClose will fail if the access key does not match. }
```

```
  FUNCTION  PCPutCh (Ch : CHAR) : BOOLEAN;
    { PCPutCh will put a character into the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if the buffer is
full or if the buffer was never opened successfully for writing. }
  FUNCTION  PCGetCh (VAR Ch : CHAR) : BOOLEAN;
    { PCGetCh will get a character from the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if there is
nothing more to read or if the buffer was never opened successfully for
reading. }
  FUNCTION  PCPutLine (L : SUStr) : BOOLEAN;
    { PCPutLine will put a string into the buffer, followed by a CR.  The
boolean result will indicate whether the operation was successful.  It will
fail if the buffer is full or if the buffer was never opened successfully for
writing. }
  FUNCTION  PCGetLine (VAR L : SUStr) : BOOLEAN;
    { PCGetLine will get a line from the buffer.  The boolean result will
indicate whether the operation was successful.  It will fail if there is
nothing more to read or if the buffer was never opened successfully for
reading. }
  FUNCTION  PCShellCmd (Cmd : INTEGER; P : SUStrP): BOOLEAN;        {ff 3/7/84}
```

QuickPort Programmer's Guide

Contents

Preface

About This Manual

This manual describes QuickPort, a set of private and intrinsic units that facilitate porting Pascal programs to the Lisa desktop. This manual is written for experienced Lisa Pascal programmers who are already familiar with the Lisa Workshop and the Lisa Operating System and who understand the concepts and conventions used by the Lisa User Interface. In addition, those who intend to write terminal emulators are assumed to know Clascal.

For material not covered in this manual, refer to one of the listed documents for additional information:

- *Operating System Reference Manual for the Lisa.*
- *Workshop User's Guide for the Lisa.*
- *Lisa Internals Manual.*
- *Lisa User Interface Guidelines.*
- *An Introduction to Clascal.*

Chapter 1
Introduction

Introduction

1.1 What is QuickPort?

QuickPort is a set of private and intrinsic units that provide a fast and reliable way to run Pascal programs in the Lisa Office System. By using QuickPort, you can make a few changes in a typical Pascal program, and it will run on the Lisa desktop. Applications that use QuickPort are integrated so that you can cut and paste to and from other Lisa applications. QuickPort also provides standard menus for all applications that use it.

1.2 Types of QuickPort Applications

Before you can use QuickPort to port your application to the Desktop, your program must

- Run in the Lisa Workshop.
- Use only **readlns** and **writelns** for text input and output.

A Pascal program that runs in the Lisa Workshop and uses **readlns** and **writelns** for text input and output is called a "vanilla" Pascal program. Vanilla Pascal programs can be ported to the desktop with very few changes.

You can also use QuickDraw calls for graphics, use the mouse to get input, and use a subset of the Lisa Hardware Interface. However, the addition of a graphic panel and use of the hardware interface involves more coding to acheive the port than a vanilla Pascal program.

1.3 Additional Features

QuickPort also provides a set of additonal procedures for configuring the panels, text output, graphic output, and for applications that use the hardware interface. Using these features, you can increase the power of your application. The additional QuickPort features are described in Chapter 3.

Chapter 2
Using QuickPort

Using QuickPort

2.1 QuickPort Program Requirements

Vanilla Pascal programs need nothing but the addition of one or two list elements to the USEs statement in its main program. A vanilla Pascal program runs in the Lisa Workshop and uses only **readlns** and **writelns** for input and output. You can use QuickDraw, but there are some minor changes required. See Section 3.4.1.1, QuickDraw Requirements, in Chapter 3, for more information. If you use the Lisa Hardware Interface, you must modify your program and use the QuickPort Hardware Interface. The QuickPort Hardware Interface is a subset of the Lisa Hardware Interface; it is described in Section 3.11, Procedures for the QuickPort Hardware Interface, in Chapter 3.

If your program is a vanilla Pascal program, you can either enhance it using the QuickPort features described in Chapter Three, or port it directly to the Lisa Desktop. If you wish to port your program to the Lisa Desktop without using any of the additional QuickPort features, make sure your program works in the QuickPort execution environment described in Section 2.3, and then turn to Chapter Four: Bringing Your Application to the Lisa DeskTop.

2.2 Choices for QuickPort Applications

You can produce several different types of applications using QuickPort:

- Applications that produce text output only.
- Applications that use QuickDraw to produce graphic and/or text output.
- Graphic applications that use the QuickPort Hardware Interface to get mouse input in the graphic panel.

QuickPort provides three panels: the text panel, the input panel, and the graphic panel. The text panel saves all text output, unless the Don't Save Buffer command is chosen from the Edit menu. Any application that produces text output only gets a text panel automatically. The input panel displays text that has not been read by the program. You can choose to have the input panel or not; the default is no input panel. Any application that produces graphic output only gets a graphic panel. Such programs can use in addition, a text panel, and/or an input panel. The default is one panel.

The text and graphic panels can both be scrolled vertically and horizontally. The panels can be enlarged and shrunk to provide different views of the output. Both panels can be split vertically and horizontally, allowing the user to see different parts of the output at the same time.

2.3 The QuickPort Execution Environment

One of the most important things to remember when using QuickPort is that the Lisa Desktop is a multiprocessing intergrated environment and you can affect the state of other applications running on the desktop if you don't keep this in mind. Be particularly careful about using functions in the QuickPort Hardware Interface, because these functions change the state of hardware, thus affecting all applications (including the desktop).

QuickPort programs can be run in the background (inactive window) when they are not waiting for input. When a program running in the background needs input, it is suspended. Programs running in the background compete with the active window for CPU time. Programs with long CPU-bound loops should use either **Yield_CPU** or **QPYield_CPU** to yield the CPU to the active window.

User actions such as pulling down the menus and clicking the mouse are processed only when your program calls call screen I/O (WRITEs and **READ**s, etc.). If you have a long CPU-bound loop, be sure to use either **Yield_CPU** or **QPYield_CPU**, so that your program will be more responsive to the user. If you have a tight loop, there is no way for the user to break out of the loop, unless the debugger is loaded and you can hit the NMI key to halt the process. Be sure to put **Yield_CPU**, **QPYield_CPU**, or **PAbortFlag** in any tight loops. Note that you must call **QPConfig** to pass an ⌘-period to your program if you need to call **PAbortFlag**. **QPConfig** is described in Section 3.6 of Chapter 3.

2.3.1 Using Operating System Calls

You can make any operating system calls, but remember that Lisa has a multiprocessing environment. Whenever a document is opened, a process may be created (tools that handle multiple documents create one process that handles one or more documents). If two documents are opened from the same tool, you have two processes running separate instances of the same program. This could result in inconsistent data if **Write_Data**s and **Read_Data**s, or **Rewrite**s and **RESET**s are performed on the same file. If this is undesirable, you should add additional code to your application to check whether the file can be opened by more than one process.

2.3.1.1 Yield_CPU

Yield_CPU gives the CPU to any other ready process, but does not handle any user actions, such as pulling down menus, and moving windows. QuickPort provides an alternative procedure, **QPYield_CPU**, that allows the user to pull down menus and move the windows around.

2.3.1.2 Make_process

If you call **make_process** in a QuickPort application, the resulting processes cannot do any screen input and output.

2.3.1.3 LDSNs (Logical Data Segment Numbers)

You cannot use a logical data segment number less than 5, or larger than 11. Note that LDSN 5 is, by default, used by the Pascal heap. If you use a

Pascal heap larger than 128K bytes, LDSN 6 and up will be used for the heap. You can use **PLInitHeap** to change the Pascal heap to a different LDSN, but make sure you don't collide with the system LDSNs.

- LDSNs 1-4 -- QuickPort
- LDSN 5 -- Default Pascal heap
- LDSN 11 -- **OPEN '-printer'**, **RESET**, or **REWRITE '-printer'**
- LDSNs 12-16 -- LisaLibraries

2.3.1.4 Terminate_Process, Kill_Process

QuickPort programs should not call **Terminate_Process** or **Kill_Process**. These calls will terminate the program, leaving the user with no chance to do anything with the output. If you need to terminate program execution, use **halt** or drop through to the end statement of your program. ***PROGRAM TERMINATED*** will appear on the screen, and the user will be able to save and put away, copy, or print.

2.3.1.5 Terminating the Program Abnormally

TrmntExceptionHandler is the standard QuickPort exception handler for abnormal termination of a program. You can write your own terminate exception handler, but you must call **TrmntExceptionHandler** immediately in your exception handler. If this call is not made, the system will hang because QuickPort will not have a chance to clean up and transfer control to the desktop manager.

2.4 The QuickPort User Interface

QuickPort provides a standard user interface for its applications that is, with the exception of a few menu commands, the same as the standard Lisa user interface. Manipulating windows and using the mouse follow the standard Lisa user interface, as do opening and closing documents.

QuickPort provides some menu commands that are different from the standard Lisa menu commands. These commands allow the user to control program execution. A standard Lisa application continuously loops to get and process events. A QuickPort program, however, may run from beginning to end. When a QuickPort program reaches its end, it will not respond to input from the keyboard, and its window will remain open to allow the user to view the output. At this stage, the QuickPort application is idle, waiting for one of the following menu commands:

- Set Aside -- Places the document (without saving) in its icon on the desktop. If the document is reopened, the application will still be idle.
- Save & Put Away -- Saves the document. The process is then terminated. If this document is opened again, the program will not run

immediately -- it is waiting for the Restart command. If the user wants to browse through the document, it is not necessary to use the Restart command. Instead, use Save & Put Away, or Set Aside.

- Restart -- Restarts program execution.

QuickPort applications are started, from the desktop, by tearing off a document from the stationery pad and opening the document.

The QuickPort menus are discussed in Appendix A.

Chapter 3
Advanced QuickPort Features

Advanced QuickPort Features

3.1 Introduction to the Features

QuickPort provides a set of features that you can use to enhance your application. The additional procedures and functions are for

- Configuring the text and graphic panels.
- Controlling text output.
- Handling graphic output using the mouse for input.
- Providing printer support.
- Using the QuickPort hardware interface.
- Making use of the terminal emulators.

You can combine any of these procedures and functions within a QuickPort application.

You can also write your own terminal emulator. To do this you must know enough Clascal to understand subclasses, methods, and overriding methods. Read *An Introduction to Clascal* before attempting to write your own terminal emulator. See Appendix B, Writing Your Own Terminal Emulator for more information.

The logical device, '**-printer**', behaves in much the same way as it does in the Workshop, but also interacts with the Desktop's print manager. A section on printer support is included in this chapter.

3.2 Text Input and the Input Panel

QuickPort programs get input in two ways: from the keyboard, and from the clipboard. The input panel displays the text that has not yet been consumed by the program. Text in the input panel comes from two sources: "type ahead" text (text which is entered from the keyboard too quickly to be echoed immediately by the program), and text from the clipboard that will be "pasted" into the text window. The 'Read Input from Clipboard' command places the selected text in the input buffer. When the program does a **read**, the text in the input buffer is read first**d**. If the input buffer is empty, the **read** waits for input from the keyboard or from a paste command.

3.3 Text Output and the Text Panel

The text output panel displays the **writeln** output from the program. The text panel corresponds to the Pascal device **output** and the logical device

'**-console**'. The text panel emulates a terminal display. The default size of the screen area is 24 lines by 80 columns. The width of the text panel can be changed either by the program, or by the user from the Setup menu. The Setup menu is described in Appendix A.

The text panel has a *buffer area* that saves text as it is scrolled above the screen area. The size of the buffer area is increased automatically as lines are saved. The size of the buffer is limited to the amount of memory available to increase the size of the buffer. When the buffer size reaches its limit, the lines scrolled off the top of the buffer area will not be saved. The limit is approximately 3500 80-character lines. The user can choose to save or not save scrolled output using the Setup menu. The Edit menu is described in Appendix A.

The screen area has a cursor that is affected by **readlns** and **writelns** from the program. The cursor position is always:

- Inside the screen area.
- Relative to the top left position of the screen area.

The cursor position is the insertion point for input. No menu commands change the logical cursor position; it is controlled solely by the program. The cursor position is always visible when there is a **read** from the program. In other words, if the panel has been scrolled so that the cursor position is hidden, QuickPort scrolls back to the cursor position when encountering a **read**. The cursor home position is the top left position of the screen area.

3.4 Graphic Output, the Graphic Panel, and Mouse Input

Graphics in QuickPort applications are created by QuickDraw. QuickPort provides an option that allows you to choose two panels, one for text output and one for graphic output, or one panel for both text and graphic output. The graphic panel corresponds to the Workshop screen. The screen size is 720 pixels wide and 364 pixels high. The entire graphic panel is equal to the screen area in the text panel. There is no buffer area in the graphic panel because graphic output will not be scrolled out of the graphic panel. All graphic objects created by the program are saved in the graphic panel using a QuickDraw picture.

In the text panel, the mouse is used to select text. In the graphic panel, mouse clicks are saved and passed to the program. Whenever the mouse button is pressed inside the graphic panel, a mouse event, **mouseDown**, with the mouse location is saved. When the mouse button is pressed while the mouse is moved, another mouse event, with different locations, is saved. When the mouse button is released, a **mouseUp** event is saved. To see if there are any mouse events in the queue, call **MouseEvent**. **MouseEvent** returns one event at a time, until there are no more mouse events in the queue. When **MouseEvent** is called, if the mouse button is down, control will not be returned to the caller until the button is released. For this

reason, **VGetMouse** should not be used after a call to **MouseEvent**, because the mouse may be moved. Each mouse event stores a mouse location indicating where the mouse button was pressed. **VGetMouse** lets you track the mouse location when the mouse button is not down.

For more information on **MouseEvent**, Refer to Section 3.8.1.3.

3.4.1 QuickDraw Requirements

Pascal programs that run in the Lisa Workshop and use QuickDraw, call QDINIT and **OpenPort** (in the QD/Support unit). To use QuickDraw you must

- Remove the call to QDINIT and **OpenPort**. QuickPort initializes QuickDraw and opens a grafPort for drawing to the graphic panel.
- Not open a picture in this grafPort since QuickPort uses a picture to save the graphic output.
- Not customize low-level QuickDraw drawing routines in this grafPort.

If your program needs to use pictures, you can open a picture in another grafPort. If your program needs to redefine any of the QuickDraw low-level routines, you can do this in another grafPort. If your application uses multiple grafPorts, you must switch to the QuickPort grafPort whenever you want to draw to the screen.

If your application calls **DrawPicture**, you must call another QuickDraw drawing routine before calling **DrawPicture**. This is because QuickPort opens the picture when the first QuickDraw drawing routine is encountered. If **DrawPicture** is the first drawing routine encountered, QuickPort's picture will be opened incorrectly because QuickPort can handle only one picture at a time. Here is an example showing how to avoid such collisions:

```
GetPort (sysportptr); {saves system port}
OpenPort (@myPort);  {references alternate port}
myPicture := OpenPicture (thePort^.portRect);

. . . make  your QuickDraw calls here . . .

ClosePicture;
SetPort (sysportptr);     {switches to system grafPort}
EraseRect (thePort^.PortRect);  {opens system picture
                           -- any drawing routing can
                           be used}
DrawPicture (myPicture, thePort^.PortRect);
```

If you call **OpenPicture** while the QuickPort grafPort is the current port, the following alert message appears on the screen and the program is aborted:

```
Your QuickPort tool has called another OpenPicture inside
the QuickPort grafPort.  This tool will be aborted.
```

The QuickDraw procedure **ScrollRect** is not supported by QuickPort. **ScrollRect** is not supported because QuickPort uses a picture to save the graphic output, and the effect of **ScrollRect** is not saved in a picture. This means that if the user scrolls the window, the picture is redrawn to the window as if **ScrollRect** had not been called.

The size limit for the QuickPort picture is 32K bytes. When the picture approaches this size, an alert is displayed. Subsequent graphic output is displayed on the screen, but is not saved in the picture. As the size of the picture increases, the redrawing that happens as the picture is scrolled or the window moved slows. You can find out the current picture size by calling **QPGrafPicSize**. Once the picture size reaches 32K bytes, the only way to save the remaining graphic output is to **EraseRect** the entire screen (**thePort^.PortRect**). The effect of this call is to delete the old picture and create a new picture.

You can draw bit images in the QuickPort grafPort. The entire graphic panel, including the bit images, can be printed. You can copy the bit images to a LisaWrite document, but you cannot copy bit images from a QuickPort application to a LisaDraw document.

3.5 Required Change to Your Program

Before you can call any of the additional QuickPort procedures, you must add **UQPortCall** to your USES list:

```
{$U QuickDraw}      QuickDraw,
{$U QP/UQPortCall}  UQPortCall,
{$U QP/UQuickPort}  UQuickPort; {or UQPortGraph,  or
                     UQPortVT100, or UQPortSoroc}
```

3.6 Procedures for all Applications

3.6.1 Configuring the Panels -- QPConfig

You can choose several different ways to orient the panels in QuickPort applications. The procedure **QPConfig** lets you rearrange the panels and their orientations. Figure 1 shows some of the different layouts.

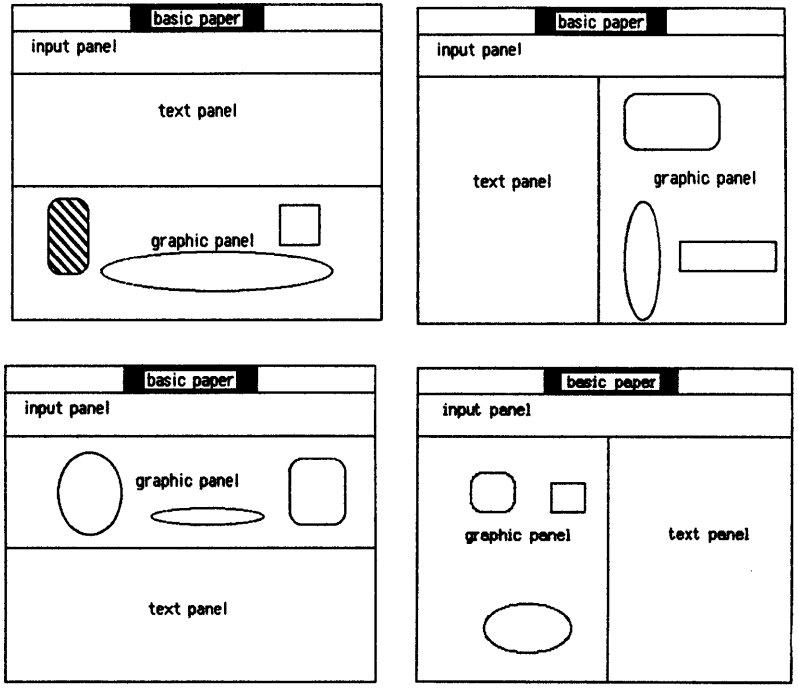


Figure 1.
QuickPort Window Layouts

Call the **QPConfig** procedure from your main program before any screen input and output is performed. You must set all the fields of a global variable of type **TQPConfigRec**.

```
PROCEDURE QPConfig (config : TQPConfigRec);
```

where

```
TQPConfigRec = RECORD
tosaveBuffer     : BOOLEAN; {save lines in
                   buffer}
passApplePeriod  : BOOLEAN; {pass apple '.' to
                   main program}
showInputPanel   : BOOLEAN; {display input
                   panel}
CASE twoPanels   : BOOLEAN OF {have both text
                   and graphic panels}
```

```
        TRUE : (vhs : VHSelect; {vertical or
                                 horizontal split.  VHSelect
                                 is defined in QuickDraw}
        grPanelSize : INTEGER); {initial width or
                                 height in pixels, if < 0,
                                 text panel is below or right
                                 of the graph panel}
END;
```

If **QPConfig** is not called, the default values are used. These defaults are in effect only if **QPConfig** is never called. If you call **QPConfig** you must set all fields, or else they will be undefined.. The default values are:

tosaveBuffer	false
passApplePeriod	false
showInputPanel	false
twoPanels	false

The graphic and text panels can be oriented in several different ways on the screen. To use **QPConfig** to set up the panels, you must first declare a variable of type **TQPConfigRec**. For example,

```
    .
    .
    .
VAR
MyConfig:          TQPConfigRec;
    .
    .
    .
QPConfig(MyConfig);
    .
    .
    .
```

To have both a graphic and a text panel, **twoPanels** must be TRUE. You must initialize the **vhs** field if you set **twoPanels** to TRUE. Once you have two panels, you can choose to split the windows on the screen vertically or horizontally. Refer to Figure 1 to see what the screen looks like with vertical and horizontal splits between windows. Then you can set the **grPanelSize** field to the size you want the graphic panel when the document is first opened (the text panel will take up the remaining space in the window).

If **QPConfig** is not called, the default values are used. Programs that handle only text output have a default of one text panel. Programs that handle graphic output have a default of one graphic panel.

3.7 Procedures for Using the Text Panel

The procedures for QuickPort applications that produce text output allow you to:

- Change the terminal parameters.
- Get raw input from the console.
- Clear the screen.
- Control the cursor.
- Set and clear tabs.
- Control keyboard input.
- Change the character style.

3.7.1 Changing the Terminal Parameters -- SetupTermPara

SetupTermPara sets the terminal parameters for the screen area in the text panel. You can call **SetupTermPara** from your terminal emulator or from your main program, but the call must be made before any screen input or output is performed. If **SetupTermPara** is not called before performing screen input or output, the default parameters will not be changed. If you call **SetupTermPara** you must set all parameters.

```
PROCEDURE SetupTermPara (termpara : TTermPara);
```

where

```
maxPosLines     = 50; {max possible lines for any
                       terminal emulator}
maxPosColumns   = 132;

Tcursorshape = (blockshape, underscoreshape,
                invisibleshape);

TTermPara = RECORD
   rowsize           : 1..maxPosLines;
   columnsize        : 1..maxPosColumns;
   toWraparound      : BOOLEAN;
   keytoStopOutput   : CHAR;
   keytoStartOutput  : CHAR;
   tmcursorShape     : Tcursorshape;
END;
```

If **SetupTermPara** is not called, the default values are used:

rowsize	24 lines
columnsize	80 columns
toWraparound	TRUE
keytoStopOutput	⌘-S
keytoStartOutput	⌘-Q
tmcursorShape	Block

3.7.2 Getting Raw Input from the Console — Vread

You can use **Vread** instead of **read** to get keyboard input and the control keys. **Vread** does not echo characters as they are **read**.

```
PROCEDURE Vread (VAR ch: CHAR; VAR keycap : QPByte;
                 VAR applekey, shiftkey,
                 optionkey: BOOLEAN);
```

The keycap is useful when you need to distinguish the numeric keypad from the main keyboard. Refer to Section 3.11.4 for the keycap definition. Note that the option key is typically used to generate extended Lisa characters. The extended Lisa characters are those characters in the range above ASCII 127. Try not to use the option key for other purposes to avoid confusing the users.

3.7.3 Clearing the Screen — ClearScreen

ClearScreen provides six different ways to clear all or part of the screen. The six ways are:

- Clear the whole screen.
- Clear from the cursor position to the end of the screen.
- Clear from the beginning of the screen to the cursor position.
- Clear the whole line.
- Clear from the cursor position to the end of line.
- Clear from the beginning of the line to the cursor position.

```
PROCEDURE ClearScreen (clearkind : INTEGER);


{clearkind definition for ClearScreen procedure}
  sclearScreen  = 1;     {clear the whole screen}
  sclearEScreen = 2      {clear to the end of the
                         screen}
```

```
sclearBScreen = 3      {clear from the beginning
                       of the screen to the cursor
                       position}
sclearLine    = 4      {clear the whole line}
sclearELine   = 5;     {clear to end of line}
sclearBLine   = 6;     {clear from the beginning
                       of the line to the cursor
                       position}
```

3.7.4 Controlling the Cursor — VGotoxy and MoveCursor

3.7.4.1 VGotoxy

VGotoxy moves the cursor to a specified position in the window.

```
PROCEDURE VGotoxy (x, y : INTEGER);
```

VGotoxy is the same as the Pascal **gotoxy**, but faster.

3.7.4.2 MoveCursor

MoveCursor moves the cursor to a position in the window *relative to the current cursor position*. **MoveCursor** allows vertical scrolling only.

```
PROCEDURE MoveCursor (scroll : BOOLEAN; xdistance,
                      ydistance : INTEGER);
```

For the **xdistance, ydistance** parameters:

- A positive value moves the cursor to the right or down.
- A negative value moves the cursor to the left or up.

If the cursor is moved down, and **scroll** is **TRUE**, the output will be scrolled up.

3.7.5 Setting and Clearing Tabs — SetTab and ClearTab

3.7.5.1 SetTab

SetTab sets a tab at a specified column, or at the current cursor position.

```
PROCEDURE SetTab (column : INTEGER);
```

SetTab sets tab at current cursor position if **column** <0.

3.7.5.2 ClearTab

ClearTab clears a tab at a specified column, or at the current cursor position.

```
PROCEDURE ClearTab (clearAll : BOOLEAN;  column : INTEGER);
```

ClearTab clears tab at current cursor position if **column** <0.

3.7.6 Controlling Keyboard Input — StopInput and StartInput

3.7.6.1 StopInput

StopInput prevents recognition of keyboard input until **StartInput** is called.

```
PROCEDURE StopInput;
```

3.7.6.1 StartInput

StartInput allows recognition of keyboard input.

```
PROCEDURE StartInput;
```

3.7.7 Changing the Character Style — ChangeCharStyle

ChangeCharStyle changes the character attributes to any style combination defined by QuickDraw.

```
PROCEDURE ChangeCharStyle (newstyle : Style);
```

3.8 Procedures for Using the Graphic Panel

The procedures for QuickPort applications that produce graphic output allow you to use the mouse to get input. These procedures are:

- Get the current mouse location.
- Test to see if the mouse button is up or down.
- Get a mouse event.
- Get either mouse or keyboard input.

3.8.1 Mouse Routines

The mouse routines listed in this section should be used instead of the ones in the Lisa Hardware Interface.

MouseEvent is a polling function. Programs may loop on **MouseEvent** to wait for mouse input. This unnecessarily takes up CPU time. Also, if the application is run in the background, **MouseEvent** will force it to run periodically, just to find out there is no mouse input, and then control is returned to the active window. This slows down the execution and user response in the active window.

WaitMouseEvent is a blocking procedure. **WaitMouseEvent** will not return to the caller until there's a mouse event, allowing user actions to be processed immediately when there are no mouse events. When a program that uses **WaitMouseEvent** is in the background, it is suspended and consequently does not take CPU time from the active window.

3.8.1.1 VGetMouse

VGetMouse returns the current mouse location in the coordinates of the current grafPort.

```
PROCEDURE VGetMouse (VAR pt : Point);
```

Point is a type defined in QuickDraw. Refer to The Lisa Pascal Reference Manual, Appendix C, QuickDraw for the definition of **Point**.

3.8.1.2 MouseButton

MouseButton returns the current state of the mouse button.

```
FUNCTION MouseButton : BOOLEAN;
```

3.8.1.3 MouseEvent

MouseEvent returns a mouse event if there is one in the queue, and returns FALSE if there is not a mouse event in the queue. A mouse event is:

- A mouse buttondown (when the user presses the mouse button).
- Mouse motion while the button is pressed.
- A mouse buttonup (when the user releases the mouse button).

Moving the mouse without pressing the mouse button is not a mouse event. When **MouseEvent** is called, if the mouse button is down, control will not be returned to the caller until the button is released.

```
FUNCTION MouseEvent (VAR aMouseEvent : TMouseEvent)
                              : BOOLEAN;
```

where

```
TMouseEvent = RECORD
mouseLoc : Point;
clicknum : INTEGER; {max 3 for triple clicks}
mouseDown, meShift, meApple, meOption :
                        BOOLEAN;
END;
```

For each mouse down event (**mouseDown** = **TRUE**), several different **mouseLoc** events may be returned in subsequent calls. These **mouseLoc** events are always ended with a mouse up event (**mouseDown** = **FALSE**).

For a double click, **MouseEvent** returns events of down, up, down, up with the **clicknum** for the second mouse down event equal to two. If the mouse button is pressed twice, but the presses do not constitute a double click, the same sequence of events is returned, but with the **clicknum** for the second mouse down event equal to one.

For a triple click, **MouseEvent** returns events of down, up, down, up, down, up, with the **clicknum** for the third mouse down event equal to three.

If the **mouseDown** field is FALSE, all other fields are meaningless.

MeShift is TRUE if the mouse button and the Shift key are depressed.
MeApple is TRUE if the mouse button and the ⌘ key are depressed.
MeOption is TRUE if the mouse button and the Option key are depressed.

3.8.1.5 WaitMouseEvent

WaitMouseEvent gets a mouse event. **WaitMouseEvent** blocks the caller until there is a mouse event in the queue.

You should use this call instead of **MouseEvent** to avoid polling and wasting CPU time. **WaitMouseEvent** also makes a program more responsive to user events such as pulling down menus, clicking in other windows, etc., when the program is waiting for mouse input.

```
PROCEDURE WaitMouseEvent (VAR aMouseEvent :
                          TMouseEvent);
```

where

```
TMouseEvent = RECORD
mouseLoc : Point;
clicknum : INTEGER; {max 3 for triple clicks}
mouseDown, meShift, meApple, meOption :
                    BOOLEAN;
END;
```

After **WaitMouseEvent** returns, a call to **MouseEvent** will get the rest of the mouse events.

3.8.1.6 WaitEvent

WaitEvent is a combination of **read** and **WaitMouseEvent**, blocking the caller until there is either keyboard or mouse input.

You should use this call instead of **MouseEvent** and **keypress** if you want both mouse and keyboard input. **WaitEvent** does not reurn input. You must call **read**, **Vread**, or **MouseEvent** depending on the value returned from the call.

```
PROCEDURE WaitEvent (VAR fromKeyboard : BOOLEAN);
```

3.8.1.7 QPGrafPicSize

QPGrafPicSize returns the size of the picture in the system grafPort.

```
FUNCTION QPGrafPicSize : INTEGER;
```

3.9 Printer Support

The printer is designated **-printer** by the Workshop. **-printer** is a logical device. To open the printer, use **reset** or **rewrite**, passing **-printer** as the file name. To send output to the printer, use **writeln** or **write**. Use **close** when you're finished sending information to the printer. **Close** lets the printshop manager know that the program is done with the printer and causes the last page to print out. If you do not call **close** after printing is finished, the printer is considered in use, and is unavailable to all other Lisa applications.

The printer is shared by all applications in the printshop. When you send something from a QuickPort application to the printer from QuickPort, you do not get immediate output. First the document is spooled to the printer queue by the printshop manager in the Lisa Office System. If there is nothing in the queue, the information comes out a page at a time. If there is something in the queue at the time of **reset** or **rewrite**, an error message is returned.

You can change the font the printer uses by calling **PrChangeFont**. The default font is 10-point, 10-pitch Century.

Paper size, printing orientation and print resolution can be changed using the Format for Printing command in the File/Print menu. Selections made using the Format for Printing command take effect only after a **reset** or **rewrite**.

The Print and Print As Is commands in the File Print menu print all the output in the selected panel.

3.10 The Terminal Emulators

QuickPort provides three terminal environments: the standard terminal, the VT100 terminal emulator, and the SOROC terminal emulator. This section summarizes the three emulators. If you want to write your own terminal emulator, go to Appendix B, Writing Your Own Terminal Emulator.

3.10.1 The Standard Terminal

The standard terminal is the terminal environment QuickPort uses unless you specify otherwise. The standard terminal provides a set of screen and cursor control functions. The standard terminal does not use escape sequences, but does interpret a set of standard control keys at output: BELL, backspace, horizontal tab, line feed, and carriage return (without line feed). Programs that use **reads** and **readlns** will have the backspace key processed automatically, i.e., the backspace key will not be passed to your program if you use **reads** and **readlns**. If your program needs to get the backspace key, use **vread** instead.

The standard Lisa applications use the ⌘-period combination to terminate long operations. QuickPort provides an option that suspends the program when the ⌘-period key combination is detected. The default is to detect the ⌘-period combination. This option is passed in **QPConfig**, which is described in Section 3.6. When a program is suspended, the user can select the Resume command to resume program execution, or the Save & Put Away command to terminate program execution.

The Setup menu (in all QuickPort applications) lets you select 80 or 132 columns per line, turn wraparound on or off, and set the tab positions.

3.10.2 The VT100 Terminal Emulator

The QuickPort VT100 terminal emulator interprets all VT100 and VT52 escape sequences, with the exception of escape sequences related to host communications. When you use the VT100 terminal emulator, the screen area

in the text panel responds to VT100 and VT52 escape sequences from **writes** and **writelns**.

The character styles supported by the QuickPort VT100 terminal emulator are bold, underline, and highlight. Since highlighted text in Lisa applications traditionally means a selection, highlighted text in the VT100 screen area will be *shadowed*. Double-height and double-width characters are not supported.

To use the VT100 terminal emulator, add

```
{$U QP/UQPortVT100} UQPortVT100;
```

to the USES list at the beginning of your main program. For more information, refer to Section 4.1, Adding the USES List Elements, in Chapter 4.

3.10.3 The Soroc Terminal Emulator

Pascal programs that run in the Lisa Workshop, and on the Apple II or Apple III, use Soroc escape sequences for output display. QuickPort provides a Soroc-compatible terminal emulator to help port these applications to the Lisa desktop. The QuickPort Soroc terminal emulator interprets all Soroc escape sequences, with the exception of those escape sequences related to display protection.

To use the Soroc terminal emulator, add

```
{$U QP/UQPortSoroc} UQPortSoroc;
```

to the USES list at the beginning of your main program. For more information, refer to Section 4.1, Adding the USES List Elements, in Chapter 4.

3.11 Procedures for the QuickPort Hardware Interface

The QuickPort hardware interface is a subset of the Lisa hardware interface. These procedures are for the mouse, the screen, the speaker, the keyboard, the timers, and date and time.

To use the QuickPort hardware interface, you must add

```
{$U QP/Hardware}  Hardware;
```

to the list elements in your program's USES statement. Refer to Chapter 4 for more information.

3.11.1 The Mouse

The mouse procedures let you

- Set the frequency at which the current mouse location is updated.
- Choose the relationship between physical and logical mouse movements.
- Count mouse movements.

3.11.1.1 Mouse Update Frequency

The mouse location is updated periodically, rather than continuously. The frequency of these updates can be set by calling **MouseUpdates**. The time between updates can range from 0 milliseconds (continuous updating) to 28 milliseconds, in intervals of 4 milliseconds. The initial setting is 16 milliseconds.

```
Procedure MouseUpdates (delay: MilliSeconds);
```

3.11.1.2 Mouse Scaling

MouseScaling enables and disables mouse scaling. **MouseThresh** sets the threshold between fine and coarse movements.

```
Procedure MouseScaling (scale:Boolean);

Procedure MouseThresh (threshold: Pixels);
```

The relationship between physical mouse movements and logical mouse movements is not necessarily a fixed linear mapping. Three alternatives are available: unscaled, scaled for fine movement and scaled for coarse movement. Initially mouse movements are unscaled.

When mouse movement is *unscaled*, a horizontal mouse movement of x units yields a change in the mouse X-coordinate of x pixels. Similiarly, a vertical movement of y units yields a change is the mouse Y-coordinate of y pixels. These rules apply irregardless of the speed of the mouse movement.

When mouse movement is *scaled*, horizontal movements are magnified by 3/2 relative to vertical movements. This is to compensate for the 2/3 aspect ratio of pixels on the screen. When scaling is in effect, a distinction is made between *fine* (small) movements and *coarse* (large) movements. Fine move- ments are slightly reduced, while coarse movements are magnified. For scaled fine movements, a horizontal mouse movement of x units yields a change in the X-coordinate of x pixels, but a vertical movement of y units yields a change of (2/3)*y pixels. For scaled coarse movements, a horizontal movement a x units yields a change of (3/2)*x pixels, while a vertical movements of y units yields a change of y pixels.

The distinction between fine movements and coarse movements is determined by the sum of the x and y movements each time the mouse location is updated. If this sum is at or below the *threshold*, the movement is considered to be a fine movement. Values of the threshold range from 0 (which yields all coarse movements) to 256 (which yields all fine movements). Given the default mouse updating frequency, a threshold of about 8 (**threshold**'s initial setting) gives a comfortable transition between fine and coarse movements.

3.11.1.3 Mouse Odometer

MouseOdometer returns the sum of the X and Y movements of the mouse since boot time. The value returned is in (unscaled) pixels. There are 180 pixels per inch of mouse movement.

```
Function MouseOdometer: ManyPixels;
```

3.11.2 The Screen

The screen procedures are used to

- Set the size of the display screen.
- Count the number of screen refreshes.
- Set the screen contrast, set automatic screen dimming.
- Set the fade delay.

3.6.2.1 Screen Size -- ScreenSize

The display screen is a *bit mapped display*. In other words, each pixel on the screen is controlled by a bit in main memory. The display has 720 pixels horizontally and 364 lines vertically, and therefore requires 32,760 bytes of main memory. The screen size may be determined by calling **ScreenSize**.

```
Procedure ScreenSize (var x: Pixels; var y: Pixels);
```

3.11.2.2 Screen Refresh Counter -- FrameCounter

The screen display is refreshed about 60 times per second. A *frame counter* is incremented between screen updates, at the vertical retrace interrupt. The frame counter is an unsigned 32-bit integer which is reset to 0 each time the machine is booted. **FrameCounter** returns this value. To minimize flickering, an application can synchronize with the vertical retraces by watching for changes in the value of this counter. The frame counter should *not* be used as a timer; use the millisecond and mircosecond timers instead.

```
Function FrameCounter: Frames;
```

3.11.2.3 Screen Contrast -- ScreenContrast, SetContrast and RampContrast

The screen's contrast level is under program control. Contrast values range from 0 to 255 ($FF), with 0 as maximum contrast and 255 as minimum. **ScreenContrast** returns the contrast setting; **SetContrast** sets the screen contrast. The low order two bits of the contrast value are ignored. The initial contrast value is 128 ($80).

```
Function Contrast: ScreenContrast;

Procedure SetContrast (contrast: ScreenContrast);
```

A sudden change in the contrast level can be jarring to the user. **RampContrast** gradually changes the contrast to the new setting over a period of about a second. **RampContrast** returns immediately, then ramps the contrast using interrupt driven processing.

```
Procedure RampContrast (contrast: ScreenContrast);
```

3.11.2.4 Automatic Screen Dimming — DimContrast and SetDimContrast

The screen contrast level is automatically dimmed if no user activity is noted over a specified period (usually several minutes). The contrast level is dimmed to preserve the screen phospher. **DimContrast** returns the contrast value to which the screen is dimmed; **SetDimContrast** sets this value. The initial dim contrast setting is 176 ($B0).

```
Function DimContrast: ScreenContrast;

Procedure SetDimContrast (contrast: ScreenContrast);
```

3.11.2.5 Automatic Screen Fading — FadeDelay and SetFadeDelay

The delay between the last user activity and dimming of the screen is under software control. **FadeDelay** returns the fade delay; **SetFadeDelay** sets it. The actual delay will range from the specified delay to twice the specified delay. The initial delay period is five minutes.

```
Function FadeDelay: MilliSeconds;

Procedure SetFadeDelay (delay: MilliSeconds);
```

3.11.3 The Speaker

The speaker routines in this section provide square wave output from the Lisa speaker.

The speaker procedures let you

- Set the speaker volume.
- Use the speaker.

3.11.3.1 Speaker Volume — Volume and SetVolume

The speaker volume can be set to values in the range 0 (soft) to 7 (loud). **Volume reads** the volume setting; **SetVolume** sets it. The initial volume setting is 4.

```
Function Volume: SpeakerVolume;

Procedure SetVolume (volume: SpeakerVolume);
```

3.11.3.2 Using the Speaker -- Noise, Silence and Beep

Noise and **Silence** are called in pairs to start and stop square wave output. **Beep** starts square wave output which will automatically stop after the specified period of time. The effects of **Noise**, **Silence** and **Beep** are overridden by subsequent calls.

Procedure Noise (waveLength: MicroSeconds);

Procedure Silence;

Procedure Beep (waveLength: MicroSeconds; duration:

Noise produces a square wave of approximately the specified wavelength. **Silence** shuts off the square wave. The minimum wavelength is about 8 microseconds, which corresponds to a frequency of 125,000 cycles per second, well above the audible range. The maximum wavelength is 8,191 microseconds, which corresponds to about 122 cycles per second.

3.11.4 The Keyboard

Three physical keyboard layouts are defined, the Old US Layout (with 73 keys on the main keyboard and numeric keypad), the Final US Layout (76 keys) and the European Layout (77 keys). Each key has been assigned a *keycode*, which uniquely identifies the key. Keycode values range from 0 to 127. Figure 2 defines the keycodes for the Final US Layout, using the legends from the US Keyboard. The Old US Layout has three fewer keys: |\, Alpha Enter, and Right Option are not on the old keyboard. The European Layout has one additional key, ><, with a keycode of $43.

Two keys on the Old US Layout generate keycodes different from the corresponding keys on the Final US Layout. To aid in compatibility, software changes the keycode for ~` from $7C to $68, and the keycode for Right Option from $68 to $4E.

Figure 2
Keycodes for "Final US Layout"

HIGH → / LOW ↓	000 0	001 1	010 2	011 3	100 4	101 5	110 6	111 7
0000 0			CLEAR		— / -	(/ 9	E	A
0001 1	DISK 1 INSERTED		-		+ / =	) / 0	^ / 6	@ / 2
0010 2	DISK 1 BUTTON		+ / ◀		| / \	U	& / 7	# / 3
0011 3	DISK 2 INSERTED		* / ▶			I	* / 8	$ / 4
0100 4	DISK 2 BUTTON		7		P	J	% / 5	! / 1
0101 5	PARALLEL PORT		8		BACKSPACE	K	R	Q
0110 6	MOUSE BUTTON		9		ALPHA ENTER	{ / [	T	S
0111 7	MOUSE PLUG		/ / ▲			} /]	Y	W
1000 8	POWER BUTTON		4		RETURN	M	~ / `	TAB
1001 9			5		0	L	F	Z
1010 A			6			: / ;	G	X
1011 B			' / ▼			" / '	H	D
1100 C			.		? / /	SPACE	V	LEFT OPTION
1101 D			2		1	< / ,	C	CAPS LOCK
1110 E			3		RIGHT OPTION	> / .	B	SHIFT
1111 F			NUMERIC ENTER			O	N	\

The keyboard procedures allow you to

- Find out the keyboard identification number.
- Find out the state of keyboard.

3.11.4.1 Keyboard Identification -- Keyboard

The Lisa supports a host of different keyboards. Each keyboard has three major attributes: manufacturer, physical *layout*, and *legends*. The chart below describes how these three attributes are combined to form a keyboard identification number. The keyboards self-identify when the machine is turned on and when a new keyboard is attached. **Keyboard** returns the identification number of the keyboard currently attached.

Function Keyboard: KeybdId;

Function Legends: KeybdId;

Keyboard identification numbers:

7	6	5	4	3	2	1	0
Manufacturer		**Layout**		**Legends**			

Manufacturer:
00 -- APD (i.e., TKC)
01 --
10 -- Keytronics

Layout:
00 -- Old US (73 keys)
01 --
10 -- European (77 keys)
11 -- Final US (76 keys)

Layout/Legends
$0F -- Old US
$26 -- Swiss-German (proposed)
$27 -- Swiss-French (proposed)
$29 -- Portuguese (proposed)
$29 -- Spanish (proposed)
$2A -- Danish (proposed)
$2B -- Swedish
$2C -- Italian
$2D -- French
$2E -- German
$2F -- UK
$3C -- APL (proposed)
$3D -- Canadian (proposed)
$3E -- US-Dvorak
$3F -- Final US

3.11.4.2 Keyboard State -- KeyIsDown and KeyMap

Low level access to the keyboard is provided through a pollable keyboard state. This state information is based on the physical keycodes defined above. **KeyIsDown** returns the position of a single specified key. **KeyMap** returns a 128-bit map, one bit for each key.

```
Function KeyIsDown (key: KeyCap): Boolean;

Procedure KeyMap (var keys: KeyCapSet);
```

A zero indicates the key is up, a one indicates down. For the mouse plug, a zero indicates unplugged, a one indicates plugged in. Certain keys are not pollable; the corresponding bits will always be zero. These keys are the diskette insertion switches, parallel port, and power switch. (The parallel port and mouse plug keys are unreliable across reboots on older hardware.)

3.11.5 The Timers

The timer procedures let you use either the microsecond timer or the millisecond timer.

3.11.5.1 The Microsecond Timer -- MicroTimer

The **MicroTimer** function simulates a continuously running 32-bit counter which is incremented every microsecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 35 minutes, and rolls over about every 70 minutes.

```
Function MicroTimer: Microseconds;
```

The microsecond timer is designed for performance measurements. It has a resolution of 2 microseconds. Calling **MicroTimer** from Pascal takes about 135 microseconds. Note that interrupt processing will have a major effect on microsecond timings.

3.11.5.2 The Millisecond Timer -- Timer

The **Timer** function simulates a continuously running 32-bit counter which is incremented every millisecond. The timer is reset to 0 each time the machine is booted. The timer changes sign about once every 25 days, and rolls over about every 7 weeks.

```
Function Timer: Milliseconds;
```

The millisecond timer is designed for timing user interactions such as mouse clicks and repeat keys. It can also be used for performance measurements, assuming that millisecond resolution is sufficient.

3.11.6 Date and Time -- DateTime, SetDateTime and DateToTime

The date and time procedures let you

- Set the current date and time.
- Find out the date and time.

The current date and time are available as a set of 16-bit integers that represent the year, day, hour, minute and second, by calling **DateTime** and **SetDateTime**. The date and time are based on the hardware clock/calendar. This restricts dates to the years 1980-1995. The clock/calendar continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the date and time also sets the time stamp described below. **DateToTime** converts a date and time to a time stamp, defined in the next section.

```
Procedure DateTime (var date: DateArray);

Procedure SetDateTime (date: DateArray);

Procedure DateToTime (date: DateArray; var time: Seconds);
```

3.11.7 Time Stamp -- TimeStamp, SetTimeStamp and TimeToDate

The current date and time are also available as a 32-bit unsigned integer which represents the number of seconds since the midnight prior to 1 January 1901, by calling **TimeStamp** and **SetTimeStamp**. The time stamp will roll over once every 135 years. Beware--for dates beyond the mid 1960's, the sign bit is set. The time stamp is based on the hardware clock/calendar. This clock continues to operate during soft power off. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the time stamp also sets the date and time described above. Since the date and time is restricted to 1980-1995, the time stamp is also restricted to this range. **TimeToDate** converts a time stamp to the date and time format defined above.

The time stamp procedures let you

- Set the time stamp.
- Convert between standard date and time and the time stamp.

```
Function TimeStamp: Seconds;

Procedure SetTimeStamp (time: Seconds);

Procedure TimeToDate (time: Seconds; var date: DateArray);
```

The current date and time are available as a set of 16-bit integers that represent the year, day, hour, minute and second, by calling **DateTime** and **SetDateTime**. The date and time are based on the hardware clock/calendar. This restricts dates to the years 1980-1995. The clock/calendar continues to operate during soft power off, and for brief periods on battery backup if the machine is unplugged. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the date and time also sets the time stamp described below. **DateToTime** converts a date and time to a time stamp, defined in the next section.

```
Procedure DateTime (var date: DateArray);

Procedure SetDateTime (date: DateArray);

Procedure DateToTime (date: DateArray; var time: Seconds);
```

3.11.7 Time Stamp -- TimeStamp, SetTimeStamp and TimeToDate

The current date and time are also available as a 32-bit unsigned integer which represents the number of seconds since the midnight prior to 1 January 1901, by calling **TimeStamp** and **SetTimeStamp**. The time stamp will roll over once every 135 years. Beware--for dates beyond the mid 1960's, the sign bit is set. The time stamp is based on the hardware clock/calendar. This clock continues to operate during soft power off. If the clock/calendar hasn't been set since the last loss of battery power, the date and time will be midnight prior to January 1, 1980. Setting the time stamp also sets the date and time described above. Since the date and time is restricted to 1980-1995, the time stamp is also restricted to this range. **TimeToDate** converts a time stamp to the date and time format defined above.

The time stamp procedures let you

- Set the time stamp.
- Convert between standard date and time and the time stamp.

```
Function  TimeStamp: Seconds;

Procedure SetTimeStamp (time: Seconds);

Procedure TimeToDate (time: Seconds; var date: DateArray);
```

Chapter 4
Bringing Your Application to the Lisa Desktop

Bringing Your Application to the Lisa Desktop

4.1 Adding the USES List Elements

Before bringing your application to the Lisa desktop you must add the required USES list elements to your MAIN program and any of your units. Depending on what kind of application you are porting, you use different USES list elements.

1. For text output only

```
{$U QP/UQuickPort}   UQuickPort;
```

2. For graphic (QuickDraw) and/or text output

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortGraph} UQPortGraph;
```

3. If you need to use Graf3D (order of list elements important)

```
{$U QuickDraw}       QuickDraw,
{$U QP/Graf3D.OBJ}  Graf3D,
{$U QP/UQPortGraph} UQPortGraph;
```

4. For graphic (QuickDraw) and/or text output, and the hardware interface

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortGraph} UQPortGraph,
{$U QP/Hardware}     Hardware;
```

5. To use the VT100 terminal emulator

```
{$U QP/UQPortVT100}  UQPortVT100;
```

6. To use the Soroc terminal emulator

```
{$U QP/UQPortSoroc}  UQPortSoroc;
```

7. If you are calling the additional QuickPort procedures (order of list elements important)

```
{$U QuickDraw}       QuickDraw,
{$U QP/UQPortCall}   UQPortCall,
{$U QP/UQuickPort}   UQuickPort;  {or UQPortGraph,
                                   UQPortVT100,
                                   UQPortSoroc}
```

UQPortCall, unlike the other units, is only an interface and contains no code.

4.2 System Configuration

This section assumes that you are using a two-ProFile system to develop your QuickPort applications. The ProFile with the office system is called "office" in this discussion, and the ProFile with the Workshop is called "workshop." In the Workshop, set the prefix to the workshop volume. If you have a Lisa 2/10 you will not need to set the prefixes as described in this section because all development will be done on one volume.

There are two different environments to consider:

- The development environment. That is, the environment you use when developing a QuickPort application. The development environment is the Workshop.
- The run-time environment. This is the environment that the QuickPort application runs in. The run-time environment is the Office System.

4.2.1 The Development Environment

When developing, you must

- Boot from the Workshop.
- From the Workshop System Manager, set the prefix to the Workshop volume.
- Place all files listed in the USES statement on the prefix volume.

You must have the following files on your prefix volume:

- **QP/UQPortCall**
- **QP/UQPortGraph**
- **QP/UQPortSaroc**
- **QP/UQPortVT100**
- **QP/UQuickPort**
- **QP/Hardware**
- **QP/Graf3D**
- **QPLib.Obj**
- **TKLib.Obj**
- **TK2Lib.Obj**
- **QP/Phrase**

The QuickPort exec file, **qp/make**, must be on the workshop ProFile.

4.2.2 The Run-Time Environment

When running a QuickPort application, you must

- Boot from the office system.
- Have all the libraries your application needs on the office system volume.
- Have **TKLib.Obj**, **TK2Lib.Obj**, and **QPLib.Obj** on the office system volume.

4.3 Generating Your Tool

To generate your tool, you must run the QuickPort exec file, **qp/make**, or customize **qp/make** to compile, assemble, and link your tool. **Qp/make** assumes all source files are in Pascal. You can customize **Qp/make** to assemble your files. **Qp/make** forces recompilation of all your application's units, compiles your application's main program, and then links your application's units with the QuickPort intrinsic units. Then **qp/make** assigns the tool name and creates the phrase file using the tool number in the file name.

Qp/make renames the object code to a file name of the form:

{T##}obj

where **##** is the tool number you specified when **qp/make** was invoked. **Qp/make** copies the phrase file to a file name of the form:

{T##}PHRASE

If your application uses other support files, such as data files, rename the files using the **{T##}** tool number as the first part of the file name, e.g.,

{T###}support

Then, whenever a user selects the tool's icon from the desktop, all the files with the **{T##}** will be copied or deleted. **Qp/make** assumes that the source files and libraries are on the prefix volume. Refer to System Configuration above for more information.

Qp/make can be invoked in two ways, depending on how many units your application has, and depending on whether you need to specify additional object files that your application does not generate but needs to link to. If your application has four or fewer units and does not need to specify additional object files for linking, **qp/make** can be invoked as follows:

Run <qp/make (mainprogram, tool##, tool volume, unita, unitb, unitc, unitd)

where

mainprogram is the filename of your application's main program.

tool ## is the tool number you want used in your application's tool name. We recommend you use

your Lisa's serial number plus an offset. Using the serial number plus an offset will prevent duplication of tool numbers among different software developers. For testing you can use any number greater than 20.

tool volume is the office disk name. The tool will be copied to the office system.

unita, unitb
unitc, unitd Up to four units for your application. If you use more than four units, use the alternate way to invoke **qp/make** as described below.

If your application has more than four units, and/or needs additional units to link against, **qp/make** can be invoked as follows:

Run <qpmake (mainprogram, tool#, tool volume, <, UnitList, OtherObjList)

where

mainprogram, tool #, and **tool volume** are the same as above.

UnitList is a file that contains the names of all your units. When you create your UnitList file, be sure to list the units in the order they should be compiled.

OtherObjList is a file that lists any object files that your application links against but you don't generate.

Refer to some QuickPort examples programs (qp sample, note, text, and so forth) on the release diskette.

4.4 Installing Your Tool

After you run **qp/make** successfully, you must install the application on the Lisa desktop. This installation process creates a tool icon and stationery pad for your tool. To install a tool you run **InstallTool** from the Workshop. After **InstallTool** is finished, when you leave the Workshop and start the Office System, your tool and its stationery pad will be on the desktop.

To install a tool, run **InstallTool** from the Workshop with the tool number you specified in **qpmake.**

Run what Program? InstallTool

The **InstallTool** program will prompt you as follows:

Please enter the name of the device your tool is on. [PARAPORT]
This is the name of your Office System ProFile.

Please enter your tool id number
Enter the tool number you specified when you ran **qp/make.**
Remember, *every tool must have a unique number.*

Does your tool create documents? (Y or N) [YES]
If you answer no, a tool like the Calculator is created. In other

words, a tool that allows only one instance of itself at a time.

Can your tool handle more than one document at a time? If you don't know, press return. (Y or N) [NO]
Some tools, such as LisaWrite, create one process that controls multiple documents. You must answer no for QuickPort tools.

The stationery opening rectangle is defaulted to 10, 40, 640, 290
These values are always the same.

Do you wish to specify a different one? (Y or N) [NO]
If you answer yes, you are prompted for the values for the size of the rectangle when a document is opened. This rectangle will be used whenever a document is opened.

Please enter the name of your tool.
Every tool has a tool number and a tool name. When you enter a tool name, the install program places the tool name in the desktop names of the tool and its stationery.

"Tool name" has been sucessfully installed in the Office System and it will appear in the disk window associated with the device.

After you've finished running the **InstallTool** program, boot the Office System. Your application's tool and stationery pad should be on the desktop. You only need to run **InstallTool** once even if you regenerate your tool. If you do regenerate it, however, the tool name in the object file will be lost, and "Tool xx" will be listed in all the alerts. To get the tool name back in the alerts, you must run **InstallTool** again.

4.5 The Icon Editor

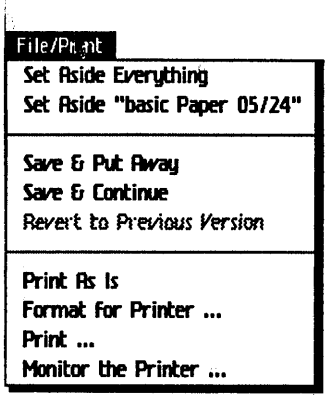
The icons created by the **InstallTool** program are blank (without pictures). If you want to design an icon for your application, contact Macintosh Technical Support.

4.6 Shipping Your Application

Your application's phrase file, as well as the object file, must be shipped. The phrase file contains the standard QuickPort menus and alerts, and it must be shipped with your application.

Appendix A
The Standard QuickPort Menus

A.1 File/Print Menu



Set Aside Everything Returns all windows to their icons without saving the contents.

Set Aside "your document" Returns the current document to its icon without saving the contents.

Save & Put Away Saves the contents of the document, closes the window, terminates the program, and returns the icon to its original location.

Save & Continue Saves the contents of the document and leaves the window open.

Revert to Previous Version Always gray -- not supported by QuickPort.

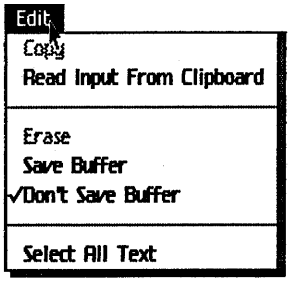
Print As Is Prints one copy of the document.

Format for Printer Sets formats in the document based on the printer that will be used.

Print Prints the document using the settings from the Format for Printer dialog box. You may choose to print multiple copies.

Monitor the Printer Shows the status of the document(s) being printed.

A.2 Edit Menu



Copy Copies the current selection onto the Clipboard. In the text panel the selection is done as in LisaWrite. In the graphic panel, the entire panel is copied. If there is a text panel, and a graphic panel, you must use Select All Graph to make the selection.

Read Input From Clipboard Places what is in the Clipboard into the input buffer.

Erase Erases the current selection.

Save Buffer Saves the lines that scroll off the top of the screen area. A check next to Save Buffer indicates that the lines will be saved.

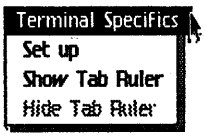
Don't Save Buffer Does not save the lines that scroll off the top of the screen area. A check next to Don't Save Buffer indicates that the lines will not be saved.

Select All Text Selects all the text in the text panel when there is a text panel.

Flush Input Clears the input panel. This command is shown only when the input panel is shown.

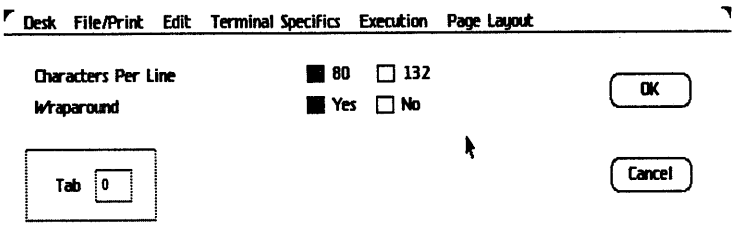
Select All Graph Selects the entire graphic panel when there is a graphic panel.

A.3 Terminal Specifics



Set up Allows you to select 80 or 132 characters per line, and line wraparound.

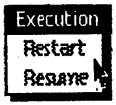
The following dialog box appears for you to fill in:



Show Tab Ruler Displays the tab ruler.

Hide Tab Ruler Hides the tab ruler.

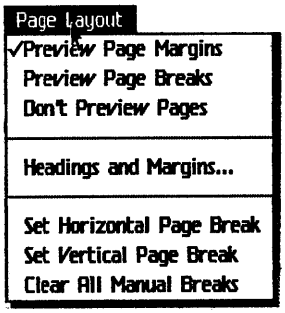
A.4 Execution



Restart Restarts program execution.

Resume Starts program execution at the point where it was suspended by an ESC-period.

A.5 Page Layout



Preview Page Margins Shows the page margins. Note that the default page margins are such that the output in the text panel will not fit in the width of an 8" by 11" page. Before printing you should adjust the left and right margins ao that each vertical page will fit in one 8" by 11" page.

Preview Page Breaks Shows the page breaks.

Don't Preview Pages Does not show the page boundaries.

Set Horizontal Page Break Sets a horizontal page break at the position of the last mouse click.

Set Vertical Page Break Sets a vertical page break at the position of the last mouse click.

Clear All Manual Breaks Clears all the page breaks set in the document.

Appendix B
Writing Your Own Terminal Emulator

Writing Your Own Terminal Emulator

B.1 Introduction

This appendix briefly discusses how to write your own terminal emulator, using the standard terminal as a template. To write a terminal emulator, you must understand Clascal. Specifically, you must understand how to extend a Clascal program by creating a subclass, overriding existing methods, and creating new methods. This section assumes you are comfortable with these basic Clascal concepts. If you don't understand Clascal, contact Macintosh Technical Support for a copy of *An Introduction to Clascal* before reading this section.

To write a terminal emulator, you create a subclass of **TStdTerm**. **TStdTerm** is the standard terminal provided by QuickkPort. The subclass you create defines the terminal emulator you want. This appendix discusses **TStdTerm**, the methods you *must* override in your subclass, and the methods used by **TStdTerm**. You can also add your own methods in your subclass.

B.2 TStdTerm

TStdTerm is the standard terminal that is used by QuickPort applications unless the VT100, Soroc, or any other terminal emulator is specified. The **TStdTerm** fields and methods are discussed in this section.

B.2.1 TStdTerm Fields

The fields you need to know about in **TStdTerm** are listed below. These fields explain how the standard terminal behaves. You may want to change some or all of this behavior in your terminal emulator.

maxLines	The maximum number of lines in the window.
maxColumns	The maximum number of columns in the window.
cursorshape	The shape of the cursor. The standard terminal uses a box cursor.
saveBuffer	To save lines as they scroll off the top of the screen into the buffer.
wraparound	BOOLEAN, whether wraparound is on or off.
stopOutputKey	Used to stop output.
startOutputKey	Used to start output.

You can only chage these fileds in your **CREATE** method.

B.2.2 TStdTerm Methods You Must Override

You must override three of these four methods in your subclass. You may want to override **CtrKeyWrite**.

B.2.2.1 CREATE

CREATE creates an object of class **TStdTerm**. You must override the **CREATE** method in your subclass.

```
FUNCTION {TStdTerm}CREATE (object: TObject; heap :
                           Theap) : TSTdTerm;
```

You must use **object** and **heap** as arguments in your **CREATE** method.

B.2.2.2 VWrite

VWrite is called by QuickPort when the program calls a **write**. You must override the **VWrite** method in your subclass to handle escape sequences that apply to your terminal.

```
PROCEDURE {TStdTerm}VWrite (VAR str : Tstr255);
```

B.2.2.3 Vread

Vread is called by QuickPort when the program calls a **read**. You must override the **Vread** method in your subclass to return any escape sequences generated from your terminal.

```
PROCEDURE {TStdTerm}Vread (VAR ach: char; VAR
                keycap : Byte; VAR applekey,
                shiftkey, optionkey ; BOOLEAN);
```

B.2.2.4 CtrKeyWrite

CtrKeyWrite handles the control keys for the terminal emulator. You should override this method in your subclass if you want to handle different control keys.

```
PROCEDURE {TStdTerm}CtrKeyWrite (ctrch: CHAR);
```

The control keys handled in the standard terminal are CR (no LF), LF, Bell, Backspace, Horizontal Tab.

B.3 Procedures Terminal Emulators Can Call

The procedures listed in this section can be called by any terminal emulators. Note that these are not methods and do not need to be overridden in your subclass.

B.3.1 Screen Control Procedures

These procedures use escape sequences.

B.3.1.1 Manipulating Lines — VGetLine and VPutLine

VGetLine deletes the specified line. **VPutLine** inserts the line at the specified line number.

```
PROCEDURE VGetLine (lineNo : INTEGER; VAR line :
                    Tstr255; delete : BOOLEAN);

PROCEDURE VPutLine (lineNo : INTEGER; VAR line :
                    Tstr255; insert : BOOLEAN);
```

B.3.1.2 Redrawing — RedrawScreen and RedrawLine

RedrawScreen and **RedrawLine** are used after **VGetLine** and **VPutLine**. **RedrawScreen** repaints the entire screen after a change to the lines or a screen size change. **RedrawLine** repaints a line after its attributes have been changed.

```
PROCEDURE RedrawScreen;

PROCEDURE VPutLine (lineNo : INTEGER);
```

B.3.1.3 Scrolling — VScrollLines

VScrollLines scrolls output on the screen without changing the data structure.

```
PROCEDURE VScrollLines (topRegion, bottomRegion :
                        INTEGER; scrollhowmanylines :
                        INTEGER);
```

A positive value for **scrollhowmanylines** scrolls down.

B.3.1.4 Changing the number of columns — ChangeMaxColumns

ChangeMaxColumns changes the maximum number of columns per line to the specified number. When **ChangeMaxColumns** is called, the corresponding character font is used. If the columns per line is 80 or less, QuickPort uses a 12-pitch font, otherwise a 20-pitch font is used.

```
PROCEDURE ChangeMaxColumns (newColumns : INTEGER);
```

B.3.1.5 Changing Fonts — ChangeFont

ChangeFont changes to the specified font. Because of cursor positioning, QuickPort supports only fixed pitch fonts.

```
PROCEDURE ChangeFont (newFont : INTEGER);
```

B.2.4 VStrWrite

VStrWrite writes the string from the cursor position. This call is the one that does the actual display of output. Terminal emulators should call this after determining there is no escape sequence in the string. This call actually displays the output. No control functions are allowed in the string. This call handles wraparound.

```
PROCEDURE VStrWrite (VAR str : Tstr255);
```

Notes