LISA

Operating System

REFERENCE MANUAL

Bill Schottstaedt

Ext. 2379

TABLE OF CONTENTS

INTRODUCTION

The Operating System is a single user system providing concurrent processes, events, exceptions, device independent I/O in a hierarchical file system, and management of code and data segmentation. This manual is intended for applications programmers who deal directly with the Operating System.

The Operating System falls naturally into four categories: file management, process management, memory management, and process communication. In each of the four chapters describing these portions of the Operating System, there is an overview of the subject that explains the terms and concepts used in the system calls. The system calls themselves are then described in some detail. A fifth chapter describes system startup procedures. The Appendices describe the Operating System interface and error codes.

CHAPTER 1

THE FILE SYSTEM

FILE OVERVIEW

INTRODUCTION

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

A file is an uninterpreted stream of eight bit bytes. A file that is stored on a block structured device resides in a catalog 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 attributes 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 meaning to the application. 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.

To handle input and output, applications do not need to know the physical characteristics of a device. Applications that do, however, can increase the I/O performance by causing file accesses on block boundaries. Each Operating System call is synchronous in that the I/O requested is performed before the call returns. The actual I/O, however, is asynchronous and is always performed in the context of an Operating System process.

To reduce the impact of an error, the file system maintains a high level of 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 program is able to discover and reconstruct damaged directories from the information stored with each file.

FILE NAMES

All the files known to the Operating System at a particular time are organized into a tree of catalogs. At the top of this tree is a predefined catalog with names for the highest level objects seen by the system. These include physical devices, such as a printer or a modem, and the volume names of any disks that are available.

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

For example,

```
    -PRINTER          names the physical printer,

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

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

    -RS232B

is illegal, but

    -RS232B-XYZ

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

RS232A	Serial Port 1
RS232B	Serial Port 2
UPPER	Upper Twiggy drive (Drive 1)
LOWER	Lower Twiggy drive (Drive 2)
DEV0, DEV6, DEV7, DEV8	Bit bucket (byte stream is flushed into oblivion)

Upper and lower case are significant in file names: 'TESTVOL' is not the same object as 'TestVol'. Any ASCII character is legal in a pathname, including the non-printing characters.

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.

DEVICES

The Lisa hardware supports a variety of I/O devices including the keyboard, mouse, clock, two Twiggy disk drives, two serial ports, a parallel port, and three expansion I/O slots. The screen, keyboard, and mouse are accessed through LisaGraf and the Window Manager. The other devices are handled by the Operating System.

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

All device calls are synchronous from the process point of view. Within the Operating System, however, I/O operations are asynchronous. The process doing the I/O is blocked until the operation is complete.

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

Serial Port 1	(RS232A)
Serial Port 2	(RS232B, the leftmost port)
I/O Slot 0	
I/O Slot 1	
I/O Slot 2	
Speaker	
10 ms system timer	
Keyboard, mouse, soft-off switch, battery powered clock	
CRT vertical retrace interrupt	
Parallel Port	
Twiggy 1	(UPPER)
Twiggy 2	(LOWER)
Video Screen	

The Operating System maintains a Mount Table which connects each available device with a name and a device number. The Device Driver associated with a device knows about the device's physical characteristics such as sector size and interleave factors for disks.

STRUCTURED DEVICES

On structured devices, such as disk drives, the File System maintains a higher level of data access built out of pages (logical names for blocks), label contents, and data clusters (groups of contiguous pages). Any file access ultimately translates into a page access. Intermediate buffering is provided only when it is needed. Each page on a structured device is self-identifying, and the page descriptor is stored with the page contents to reduce the destructive impact of an I/O error. The eight components of the page descriptor are:

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

Each structured device has a Media Descriptor Data File (MDDF) which describes the various attributes of the media 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 bitmap of which pages on the media 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. The file data need not be allocated in contiguous pages.

THE VOLUME CATALOG

On a block structured device, the volume catalog provides access to the files. The catalog is itself a file which maps user names into the internal files 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 or last modified
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 media, and the Operating System may even move it around occassionally to avoid wear on the media.

LABELS

An application can store its own information about file attributes in an area called the file label. The label allows the 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 488 bytes.

LOGICAL AND PHYSICAL END OF FILE

A file contains some number of bytes recorded in some number of physical blocks. Additional blocks might be allocated to the file, but not contain any file data. There are, therefore, two measures of the end of the file called the logical and physical end of file. The logical end of file (LEOF) is a pointer to the last stored byte which has meaning to the application. The physical end of file (PEOF) is a count of the number of blocks allocated to the file.

In addition, each open file in each process has a pointer associated with it called the file marker that points to the next byte in the file to be read or written. When the file is opened, the file marker points to the first byte (byte number 0). The file marker can be positioned implicitly or explicitly using the read and write calls. It cannot be positioned past LEOF, however, except by a write operation that appends data to a file.

When a file is created, an entry for it is made in the catalog specified in its pathname, but no space is allocated for the file itself. When the file is opened by a process, space can be allocated explicitly by the process, or automatically by the operating system. If a write operation causes the file marker to be positioned past the Logical End Of File (LEOF) marker, LEOF and PEOF are automatically extended. The new space is contiguous if possible, but not necessarily adjacent to the previously allocated space.

FILE ACCESS

There are several modes in which an application can perform input, output, or device control operations. Applications are provided with a device independent byte stream interface. 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 seen by the application, except that devices that do not support positioning can only perform sequential operations.

Applications that know the block size for structured devices can optimize performance by performing I/O on block boundaries in integral block multiples. This mode bypasses the buffering of parts of blocks that the system normally performs. Data transfers take place directly between the device and the computer memory. Although data transfers occur in physical units of blocks, the file marker still indicates a byte position in the file.

A file can be open for access simultaneously by multiple processes. All write operations are completed before any other access to the file is permitted. When one process writes to a file the effect of that write is immediately visible to all other processes reading the file. The other processes may, however, have accessed the file in an earlier state and not be aware of the change until the next time they access the file. It is left up to the applications to insure that processes maintain a consistent view of a shared file.

Each time a 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 uses the refnum in subsequent calls to refer to the file. Each operation using the refnum affects only the file marker associated with it. The refnum is global only if a process has opened the file with global access. The LEOF and PEOF values, however, are always global attributes of the file, and any change to these values is immediately visible to all processes accessing that file.

Processes can share the same file marker. In this access mode (global access) each of the processes uses the same refnum for the file. When a process opens a file in global access mode, the refnum it gets back can be used by any process. Note that [Global_Access] access allows the same file to be opened globally by any number of processes, creating any number of simultaneously shared refnums. [Global_Access,Private] access opens a file for global access. but allows no other process to Open that file. Applications must be aware of all the side effects that global accesses cause. For example, processes making global accesses to a file cannot make any assumptions about the location of the file marker from one access to the next.

Even if the access mode is not global, more than one process can have the same file open simultaneously. Each process, in this case, has its own refnum and file marker. A write operation to the file, however, is immediately visible to all readers of that file.

PIPES

Because the Operating System supports multiple processes, a mechanism is needed for interprocess communication. This mechanism is called a pipe. A pipe is very similar to any other object in the file system -- it is named according to the same rules, and can have a label. A pipe also implements a byte stream that queues information in a first-in-first-out manner for the pipe reader. Unlike a file, however, a pipe can have only one reader at a time, and once data is read from a pipe it is no longer available in the pipe.

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

When a pipe is created, its physical size is 0 bytes. You 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 is written or until all the writers close the pipe. Similarly, if the write pointer catches up with the read pointer, a writing process blocks until the pipe reader frees up some space or until the reader closes the pipe. Because pipes have this structure, there are certain restrictions on some operations when dealing with a pipe. These restrictions are discussed below under the relevant file system calls.

For massive data transfers, it is recommended that shared files or data segments be used rather than pipes.

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

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

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

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

```
        Ecode:      Error indication
        Path:       Full 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 in the catalog given in pathname. If the pathname specified in Path does not specify a volume name, the working directory is used. Label_size specifies the initial size in bytes of the label that the application wants to maintain for the object. It must be less than or equal to 488 bytes. The label can grow to contain up to 488 bytes no matter what its initial size is. Any error indication is returned in Ecode. An object cannot be created in the root catalog.

In the example below, we check to see whether the specified file exists before opening it. Applications that use the Window Manager must use a dialog box, rather than READ and WRITE.

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

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

        Ecode:    Error indicator
        Path:     Full name of object to be deleted
```

KILL_OBJECT deletes (removes) the entry given in path from the file system. Objects in the root catalog and 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 cannot be overridden.

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

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

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

        Ecode:          Error indicator
        Path:           Object's old (full) name
        Newname:        Object's new (partial) name
```

RENAME_ENTRY changes the name of an object in the file system. Newname is not a full pathname, but the new name for the object identified by Path. That is,

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

renames FORMATTER.LIST to NEWFORMAT.TEXT. The new file's full pathname is '-LISA-NEWFORMAT.TEXT'.

Predefined names in the root catalog cannot be renamed, but volume names can be renamed by specifying only the volume name in Path.

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

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

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 filename component:

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

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

When LOOKUP is called for a file system object (not a device or volume), the refnum field and all the record fields that follow that field contain invalid data. Use INFO to get this information.

The fs_info record is defined as:

```
    Uid = INTEGER;
    Info_Type = (device_t, volume_t, object_t);
    Devtype = (diskdev, pascalbd, seqdev, bitbkt, non_io);
    Filetype = (undefined, MDDFFile, rootcat, freelist, badblocks,
                sysdata, spool, exec, usercat, pipe, bootfile,
                swapdata, swapcode, ramap, userfile, killedobject);
    Entrytype = (emptyentry, catentry, linkentry, fileentry, pipeentry,
                 ecentry, killedentry);
    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;
                freecount: LONGINT;
                DTVC,                          (* Date Volume Created *)
                DTVB,                          (* Date Volume last Backed up *)
                DTVS:      LONGINT;
                Machine_id,
                overmount_stamp,
                master_copy_id: LONGINT;
                privileged,
                write_protected: BOOLEAN;
                master,
                copy,
                scavenge_flag: BOOLEAN);
            object_t:
                (size:     LONGINT; (*actual no of bytes written*)
                psize:     LONGINT; (*physical size in bytes*)
                lpsize:    INTEGER; (*Logical page size in bytes*)
                ftype:     filetype;
                etype:     entrytype;
                DTC,                        (* Date Created *)
                DTA,                        (* Date last Accessed *)
                DTM,                        (* Date last Mounted *)
                DTB:       LONGINT;         (* Date last Backed up *)
                refnum:    INTEGER;
                fmark:     LONGINT;         (* file marker *)
                acmode:    mset;            (* access mode *)
                nreaders,
                nwriters,
                nusers:    INTEGER;
                fuid:      uid;             (* unique identifier *)
                eof,
                safety_on,                  (* safety switch setting *)
                kswitch:   BOOLEAN;
                private,
                locked,
                protected:BOOLEAN);
        END;
```

The EOF field of the fs_info record is set after an attempt to write when no disk space is available, and after an attempt to read more bytes than are available from the file marker to the logical end of file. If the file marker is at the 20-th byte of a 25 byte file, you can read 5 bytes without setting EOF, but if you try to read 6 bytes, you get 5 bytes of data and EOF is set.

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

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

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

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

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

```
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 accesses
```

Before a process can perform I/O operations upon an object in the file system, it must OPEN that object. Path must specify either a pipe, device, or file. OPEN returns refnum to the process, and the process subsequently uses refnum for I/O and control operations on the open file. The manip parameter specifies the kind of access the process wants to the file: DRead, DWrite, Append, Global_Access, or Private. [DWrite] access is equivalent to [Dwrite,Append] access. If a process wants exclusive access to an object (a printer, for example), it must specify [Private] as its access mode.

If the object opened already exists and the process calls WRITE_DATA without specifying Append access, the object is 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 open for writing by two separate processes simultaneously. If the processes do not share a global refnum, they must coordinate their file accesses so as to avoid overwriting each other's data. To do this, both processes can, for example, open the file with [Append] access.

```
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 and other processes have the file open, refnum remains valid for these processes, and other processes can open the file using refnum even though a CLOSE_OBJECT call has been made against it.

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

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

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

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

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

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

Mode can be absolute, relative, or sequential. In absolute mode, offset specifies an absolute byte of the file. In relative mode, it specifies a byte relative to the file marker. In sequential mode, the offset is ignored (it is assumed to be zero) and transfers occur relative to the file marker. Sequential mode (which is a special case of relative mode) is the only allowed access mode for reading or writing data in pipes. 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, however, might not be contiguous with the previous blocks. You can use ALLOCATE to ensure physical contiguity before writing past PEOF.

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

The following program copies a file:

```
PROGRAM CopyFile;
USES (*$U Source:Syscall.Obj*) SysCall;
TYPE Byte=-128..127;
VAR OldFile,NewFile:PathName;
    OldRefNum,NewRefNum:INTEGER;
    BytesRead,BytesWritten:LONGINT;
    ErrorCode:INTEGER;
    Response:CHAR;
    Buffer:ARRAY [0..511] OF Byte;
BEGIN
WRITE('File to copy: ');
READLN(OldFile);
OPEN(ErrorCode,OldFile,OldRefNum,[DRead]);
IF (ErrorCode<>0) THEN
  BEGIN
  WRITELN('Error ',ErrorCode,' while opening ',OldFile);
  EXIT(CopyFile);
  END;
WRITE('New file name: ');
READLN(NewFile);
MAKE_FILE(ErrorCode,NewFile,0);
OPEN(ErrorCode,NewFile,NewRefNum,[DWrite]);
REPEAT
  READ_DATA(ErrorCode,
            OldRefNum,
            ORD4(@Buffer),
            512,BytesRead,Sequential,0);
  IF (ErrorCode=0) AND (BytesRead>0) THEN
    WRITE_DATA(ErrorCode,
               NewRefNum,
               ORD4(@Buffer),
               512,BytesWritten,Sequential,0);
UNTIL (BytesRead=0) OR (BytesWritten=0) OR (ErrorCode<>0);
IF (ErrorCode<>0) THEN
  WRITELN('File copy encountered error ',ErrorCode);
CLOSE_OBJECT(ErrorCode,NewRefNum);
CLOSE_OBJECT(ErrorCode,OldRefNum);
END.
```

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

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

Ecode:	Error indicator
Path:	Name of object containing the label
Label_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 label_addr, and actual is the actual number of bytes transferred. An error is returned if you attempt to read more bytes than were available in the label. You can read up to the maximum number of bytes written to the label, but cannot write more than 488 bytes to it.

```
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 sends a device-specific control request to the device driver for the device named by path. Path must name an object in the root catalog. The record dctype is defined:

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

dcVersion:	version number of format for application to driver data
dcCode:	control code for device driver
dcData:	specific control data parameters

```
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 a disk file. If possible, ALLOCATE adds count blocks to the space available to the file 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 block structured devices and pipes. 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 obviously cause the reader to read invalid and uninitialized data, so the File System returns an error in this case.

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

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

COMPACT deallocates any blocks after the block that contains the logical end of file for the file referenced by refnum. (See Figure 3 below). COMPACT applies only to block structured devices and pipes. As is the case with ALLOCATE, compaction of a pipe is legal only if the read pointer is less than or equal to the write pointer. If the write pointer has wrapped around, but the read pointer has not, compaction could destroy data in the pipe, so the File System returns an error in this case.

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

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

TRUNCATE sets the logical end of file indicator to the current position of the file marker. Any file data beyond the file marker is lost. TRUNCATE applies only to block structured devices and pipes. Truncation of a pipe can destroy data that has been written but not yet read. As the diagram shows, TRUNCATE does not change PEOF, only LEOF.

```
                                                +<--Compact--+
                                                |            |
                                                v            v
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
                  ^                 ^                        ^
                  |                 |                        |
             File Marker           LEOF                     PEOF
                  |                 |
                  +<---Truncate-----+
```

The Relationship of COMPACT and TRUNCATE

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

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

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

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

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

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

Each object in the file system has a "safety switch" to prevent costly accidents. 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 which are sharing a file should cooperate with each other when setting or clearing the safety switch.

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

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

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

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

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

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

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

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

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

```
MOUNT   (Var Ecode:Integer;
         Var VName:E_Name;
         Var Password, Device:E_Name
         Var devName:E_Name)

UNMOUNT (Var Ecode:Integer;
         Var VName:E_name)

        Ecode:     Error indicator
        VName:     Volume name
        Password:  Password for device
        Devname:   Device name
```

MOUNT and UNMOUNT handle access to block structured devices. If the password given matches the password for the volume found on the device specified, MOUNT creates an entry in the root catalog which logically attaches that volume's catalog to the file system. The name of the volume mounted is returned in the parameter vname.

UNMOUNT removes the specified volume from the root catalog, thereby removing its subtree from the file system. Nothing 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.

VName can be a device name ('RS232B' or 'DEV8', for example). In the UNMOUNT call, VName can also be a volume name without the preceding dash ('TESTVOL', not '-TESTVOL').

CHAPTER TWO

PROCESSES

PROCESSES

INTRODUCTION

A process is a piece of executable code that can be run at the same time as other processes. Although processes can share code and data, each process has its own stack. In most systems, including the one supported by the Operating System, the parallel or concurrent execution of the processes is simulated by using re-entrant code and a scheduler. The scheduler allows each process to run until some condition occurs. At that time, the state of the running process is saved, and the scheduler looks at the pool of ready-to-run processes for the next one to be executed. When the first process later resumes execution, it merely picks up where it left off in its execution.

The status of a process depends on its scheduling state, execution state, and memory state. The memory manager handles the process memory state. If any code or data segments need to be swapped in for the process to execute, the memory manager is called before the process is launched by the scheduler.

The process execution state depends on whether the process is executing in user mode or in system mode. In system mode, the process executes Operating System code in the hardware domain 0. In user mode, the process executes user code in domains 1, 2, or 3.

The process scheduling state has four possibilities. The process is "running" if it is actually engaging the attention of the CPU. If it is ready to continue execution, but is being held back by the scheduler, the process is said to be "ready". When it has completed its task and has exitted its outer block, it is "terminated". A process can also be "blocked". In the blocked state, the process is ignored by the scheduler. It cannot continue its execution until something causes its state to be changed to "ready". Processes commonly become blocked while awaiting completion of I/O. Certain Operating System calls distinguish between a process that is blocked by an I/O operation, and a process that is blocked because it has been suspended by some other process.

PROCESS STRUCTURE

A process is a program. It can use up to 7 data segments and 116 code segments simultaneously. When a process is instantiated, the Operating System creates a Process Control Block (PCB) for it. The PCB contains the process state, global id, and a pointer to a record of the process's current needs. These include pointers to its code and data segments, its stack, an area to save registers, and so on. When a process calls the Operating System, the data segments and stack of the process are remapped into domain 0 where the Operating System executes. The address space layout of system and user processes is set up to make this remap as efficient as possible:

PROCESS ADDRESS SPACE LAYOUT

```
      User Mode                               System Mode

 Seg#                                   Seg#
        +--------                               +--------
  0     | Unavailable                    0      | Low memory (512 Read-Only bytes)
        +--------                               +--------
  1     | User Code Segments             1      | OS Code Segments
  .     |                                .      |
  .     |                                .      |
  .     |                                .      |
        |                                       |
        |                                       +--------
        |                               95      | Real Memory Access (I/O Space)
        |                                .      | (16 needed for 2 megabyte access)
        |                                .      |
        |                               111     |
        |                                       +--------
        |                               112     | Supervisor Stack
        |                                       +--------
        |                               113     | System Jump Table
        |                                       +--------
        |                               114     | Sysglobal data
        |                                       +--------
        |                               115     | SysLocal of currently executing process
        +--------                               +--------
  116   | LDSN 1                        116     | User Data Space
   .    |                                       |
   .    |                                       |
   .    |                                       |
        |                                       |
  122   | LDSN 7                                |
        +--------                               |
  123   | Stack                                 |
        +--------                               |
  124   | Shared Intrinsic Unit Data            |
        +--------                               |
  125   | I/O Space                             |
        +--------                               |
  126   | Reserved                              |
        +--------                               |
  127   | Screen                        127     | Screen
        +--------                               +--------
```

During execution, the process stack is:

```
                  PROCESS STACK LAYOUT

    High Memory        +-------------------------+
                       | Caller's stack frame    |
                       .                         .
                       .                         .
                       | Caller's dynamic link   |<- - +
                       .                         .     |
                       .                         .     |
                       +-------------------------+     |
                       | Function Result (only   |     |
                       | for a function)         |     |
                       +- - - - - - - - - - - - -+     |
                       | Procedure arguments     |     |
                       +- - - - - - - - - - - - -+     |
                       | Static Link (only for a |     |
                       | level 2 or higher proc) |     |
                       +- - - - - - - - - - - - -+     |
                       | Return Address          |     |
                       +- - - - - - - - - - - - -+     |
         (A6) - - - ->| Dynamic Link            |- - -+
                       +- - - - - - - - - - - - -+
                       | Local frame             |
                       +- - - - - - - - - - - - -+
                       | Dynamic requirements    |
    Low Memory         +-------------------------+<- - - (A7)
```

Each process has an associated priority, an integer between 1 and 255. The process scheduler usually executes the highest priority ready process. The higher priorities (200 to 255) are reserved for Operating System and Filer processes.

PROCESS HIERARCHY

When the system is first started, several system processes exist. At the base of the process hierarchy is the root process which handles various internal Operating System functions. It has at least three sons, the memory manager process, the timer process, and the shell process. The memory manager process handles code and data segment swapping. The shell process is a simple command interpreter which you can use to run programs and create other processes. In the final Lisa system, the shell process will be the Filer. The timer process handles timing functions such as timed event channels.

```
                     Root Process
                       /  |  \   \
                      /   |   \   \-----------+
                     /    |    \              |
                    /   Shell   \          Other...
              +----/   Process   \----+
              |           |           |
        Memory Manager    |         Timer
          Process       User       Process
                       Process
                       /  |  \
                      /   |   \
                  Other User Processes
```

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

PROCESS CREATION

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

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. Process handling calls are not allowed on Operating System processes.

PROCESS SCHEDULING

Process scheduling is based on the priority established for the process. The system usually attempts to execute the highest priority ready process. Once it is executing a process loses the CPU only under the following conditions:

* The running process becomes blocked (during I/O, for example).

* The running process lowers its priority below that of another ready process or sets another process's priority to be higher than its own.

* The running process yields the CPU to another process.

* The running process activates a higher priority process or suspends itself.

* The running process makes any Operating System call when a higher priority ready process exists.

* The running process causes code to be swapped or its stack to be expanded.

Because the Operating System currently 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 its pool of ready processes. If the new process requires code or data to be swapped in, 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.

PROCESS TERMINATION

A process terminates when it hits its 'END.' statement, when it calls TERMINATE_PROCESS, when some process calls KILL_PROCESS on it, when its father process terminates, or when it runs into an abnormal condition. When a process terminates, a "terminate" exception condition is signalled on the calling process and all of the processes it has created. A process can declare an exception handler for this condition to insure that its house is in order before its demise.

Termination involves the following steps:

1. Signal the SYS_TERMINATE exception on the current process.

2. Execute the user's exception handler (if any).

3. Send the SYS_SON_TERM event to the father of the current process if a local event channel exists.

4. Instruct all sons of the current process to terminate.

5. Close all open files, data segments, and event channels.

6. Wait for all the sons to finish termination.

7. Release the PCB and return to the scheduler.

A process can protect itself from termination by disabling the "terminate" exception. Under normal circumstances, however, a process should cooperate with the Operating System by viewing the terminate exception as an opportunity to clean up its act before it is terminated. If a process disables the terminate exception and then, illogically, calls TERMINATE_PROCESS, the Operating System forces the process to terminate.

PROCESS SYSTEM CALLS

```
MAKE_PROCESS (Var ErrNum:Integer;
              Var Proc_id:LongInt;
              Var ProgFile:Pathname;        (* PathName = STRING[255] *)
              Var EntryName:NameString;     (* NameString = STRING[20] *)
              Evnt_chn_refnum:Integer);
```

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

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

Evnt_chn_refnum is an event channel supplied by the calling process (event channels are discussed later). The Operating System uses the event channel identified by evnt_chn_refnum to send the calling process events regarding the created process (for example, SYS_SON_TERM). If evnt_chn_refnum is zero, the calling process is not informed when such events are produced.

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

Any error encountered during process creation is reported in ErrNum.

The following example uses Operating System calls that have not been fully discussed yet. It should, however, provide an example of process and event management.

```
PROCEDURE ExecuteProgram;
CONST CannotOpenProgFile=130;    (* error returned by MAKE_PROCESS *)
VAR PName:PathName;              (* pathname of program to execute *)
    Null_Entry:NameString;       (* null entry point name *)
    ErrorCode:INTEGER;           (* Error return for system calls *)
    Son_Id:LONGINT;              (* id of new process for program *)
    ec_refnum:INTEGER;           (* returned by WAIT_EVENT_CHN *)
    term_event:r_eventblk;       (* 'SYS_SON_TERM' event block *)
    event_ptr:p_r_eventblk;      (* pointer to term event *)
    comm_chan:INTEGER;           (* refnum of communication channel for sons *)
    Son_Wait_List:t_waitlist;    (* record for WAIT_EVENT_CHN *)
    null_ec:PathName;            (* null exception pathname for OPEN_EVENT_CHN *)
    null_excep:t_ex_name;        (* null exception name *)
BEGIN
Null_Entry:='';                  (* alternate entry points are currently no-ops *)
null_ec:='';
null_excep:='';
event_ptr:=@term_event;
WriteDialog('Execute what file? ');
                                 (* WriteDialog opens a dialog box, sets its
                                    height, and writes the string in it *)
ReadDialog(pname);               (* ReadDialog gets the program file name from
                                    the dialog box using EventAvail and
                                    GetNextEvent supplied by the Window Manager *)
IF (pname<>'') THEN              (* if pname is null, quit *)
  BEGIN
  OPEN_EVENT_CHN(ErrorCode,Null_ec,Comm_Chan,Null_excep,TRUE (* receive *));
                                 (* set up communication channel for process
                                    that will run the program pname *)
  WITH Son_Wait_List DO
    BEGIN
    Length:=1;
    refnum[0]:=Comm_Chan;
    END;
  MAKE_PROCESS(ErrorCode,Son_id,pname,Null_Entry,comm_chan);
  IF (ErrorCode=CannotOpenProgFile) THEN
    WriteDialog(CONCAT(pname,' not found.'));
  SETPRIORITY_PROCESS(ErrorCode,MY_ID,1);
                                 (* wait at low priority for son to terminate *)
  WAIT_EVENT_CHN(ErrorCode,Son_Wait_List,ec_refnum,event_ptr);
  SETPRIORITY_PROCESS(ErrorCode,MY_ID,200);
                                 (* return to normal priority *)
  END;
END;
```

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

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

The life of a process is ended by TERMINATE_PROCESS. This call causes a "terminate" exception to be signalled on the calling process and on all of the processes it has created. The process can declare its own "terminate" exception handler to handle whatever cleanup it needs to do before it is completely terminated by the system. When the terminate exception handler is entered, the exception information block contains an integer 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

If the terminating process was created with a communication channel, event-ptr points to the event text information that the Operating System sends to the process's creator. The event type in this case is SYS_SON_TERM.

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

```
    CONST size_etext = 9;  (* event text size - 40 bytes *)

    TYPE t_event_text = ARRAY [0..size_etext] OF LongInt;
         s_eventblk = t_event_text;
```

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

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

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

A process can call INFO_PROCESS to get a variety of information about any process known to the Operating System. Use the function My_Id to get the Proc_id of the calling process. ProcInfoRec is defined as:

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

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

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

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

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

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

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

* Executing in user mode.

* Stopped due to a SUSPEND_PROCESS call.

* Stopped due to a DELAY_TIME call.

* Stopped due to a WAIT_EVENT_CHN or SEND_EVENT_CHN call, or READ_DATA or WRITE_DATA to a pipe.

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

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

SUSPEND_PROCESS allows a process to suspend (block) any other 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, and by the timer (see DELAY_TIME), and for many other reasons.

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

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

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

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

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

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

SETPRIORITY_PROCESS changes the scheduling priority of the process referred to by proc_id to new_priority. The higher the priority value (which must be between 1 and 255), the more likely the process is to be allowed to execute. Because Operating System processes execute with priorities between 200 and 250, it is suggested that applications execute at lower priorities.

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

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

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

MY_ID

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.

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

sets the priority of the calling process to 100.

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

```
PROGRAM Father;
USES (*$U Source:SysCall.Obj*) SysCall;
VAR ErrorCode:INTEGER;          (* error returns from system calls *)
    proc_id:LONGINT;            (* process global identifier *)
    progname:Pathname;          (* program file to execute *)
    null:NameString;            (* program entry point *)
    Info_Rec:ProcInfoRec;       (* information about process *)
    i:INTEGER;
    Answer:CHAR;
BEGIN
ProgName:='SON.OBJ';            (* this program is defined below *)
Null:='';
MAKE_PROCESS(ErrorCode,Proc_Id,ProgName,Null,0);
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.
```

CHAPTER 3

MEMORY MANAGEMENT

MEMORY MANAGEMENT OVERVIEW

INTRODUCTION

Each process has a set of code and data segments which must be in physical memory during execution of the process. The transformation of the logical address used by the process to the physical address used by the memory controller to access physical memory is handled by the memory management unit (MMU).

A LIMITED HARDWARE PERSPECTIVE

Addresses in LISA have three parts: a domain (context) number, a hardware segment number, and an offset. A hardware segment is a contiguous logical address space with a distinct address protection. The hardware mapping registers determine each hardware segment's type, length (in pages of 512 bytes), and origin in physical memory. The segment type (ReadOnly, ReadWrite, or Stack) controls access to that segment.

Each segment can have up to 128 Kbytes of memory. The Operating System provides data segments larger than 128 Kbytes by allocating adjacent MMU registers to a single logical segment. 128 segments are mapped by a single domain, so each of the four domains provides a cache of an entire segment map. The Operating System runs in domain 0; application programs can operate in domains 1, 2, or 3. The use of domains speeds up process switching.

DATA SEGMENTS

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

A process can require additional data segments for such things as heaps and process to process communication. These added requirements are made known to the Operating System at run time. 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 process.

The 68000 hardware requires that all data segments that are part of the process's working set be in physical memory and mapped by hardware segment registers during execution of the process. It is the responsibility of the process to ensure that this requirement is met.

THE LOGICAL DATA SEGMENT NUMBER

Besides the stack segment, a process can have up to seven data segments in its working set at any given time. Other data segments can be available to the process, but not actually be members of the working set. To inform the Operating System that it wants a certain data segment to be available, the process associates that segment with a "logical data segment number" (LDSN). When the process wants the data segment placed in memory and made a member of the working set, it "binds" that segment to its associated LDSN. The LDSN, which has a valid range of 1 to 7, is local to the calling process. The process uses the LDSN to keep track of where a given data segment can be found. More than one data segment can be associated with the same LDSN, but only one such segment can be bound to an LDSN at any instant and thus be a member of the working set of the process.

SHARED DATA SEGMENTS

Cooperating processes can share data segments. The segment creator assigns the segment a unique name (a file system pathname). All processes that want to share that data segment must then use the same segment name. If the shared data segment contains address pointers to segments, then the cooperating processes must also agree upon a common LDSN to be associated with the segment. This LDSN is transformed by the Operating System into a specific mapping register, so all logical data addresses referencing locations within the data segment are consistent for all processes sharing the segment.

As an example of the use of shared data segments, consider the following situation: a process creates five other processes and wants to use a different data segment for communication with each of them. The process can associate and bind the five data segments with LDSN values 1 to 5. Since it can access all five segments at will, this method can have performance advantages, but all five data segments must be in memory during execution. If on the other hand, the process associates all five data segments with the same LDSN, only one such segment must be in memory at any time, but the process must bind and unbind the segments to the LDSN whenever a specific segment is needed. The application designer must weigh the advantages and disadvantages of each method for the application being developed.

PRIVATE DATA SEGMENTS

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

Suppose a process has a data segment already bound to LDSN 2. If the program tries to bind a 256 Kbyte data segment to LDSN 1, the Operating System returns an error because the 256 Kbyte segment needs two consecutive free LDSN's. Instead, the program should bind the segment to LDSN 3 and the system implicitly also uses LDSN 4. If the program has no bound LDSN's, it can start its heap segment at LDSN 1, and as the heap grows, it can expand upward through the 7 LDSN's.

CODE SEGMENTS

Division of a program into multiple code segments (swapping units) is dictated by the programmer. If a program is so divided, the Linker creates a jump table to insure that intersegment procedure references are handled properly. The MMU registers can map up to 116 code segments. The allocation of the register numbers is given in the Process Structure section of the Process chapter.

A JSR, RTS, or JMP.L to a non-resident code segment causes a bus error which results in a trap to the Operating System (a software implementation of absence traps). The Operating System brings the code segment into physical memory and returns control to the process, allowing the procedure reference to continue.

THE PROCESS STACK

Because the Operating System sometimes needs to scan the stack of a process, certain conventions must be observed:

* Register A7 is the stack pointer of the process.
* Register A6 is the link register for the process stack.
* All procedures must execute the LINK instruction using A6 as the link register before any local data is placed on the stack or another procedure call is executed.

These conventions are obviously hidden from the programmer's view in high level languages, but must be followed by assembly language programmers.

Stack expansion is handled automatically by the Operating System.

SWAPPING

When a process executes, the following segments are required to be in physical memory and mapped by mapping registers:

* The current code segments being executed

* All the data segments in the process working set.

The Operating System insures that this minimum set of segments is in physical memory before the process is allowed to execute. If a required segment is 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.

MEMORY MANAGEMENT CALLS

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

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	Pointer to contents of data segment
Ldsn:	Logical data segment number

MAKE_DATASEG creates the data segment identified by the pathname, segname, and opens it for immediate read-write access. Segname is a true file system pathname. If segname is null, the data segment can be accessed only by the calling process; otherwise, the segname allows the segment to be shared with any process in the system.

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

Disk_size determines the number of bytes of swapping space to be allocated to the segment on disk. If Disk_size is less than Mem_size, the segment cannot be swapped out of main memory. In this case the segment is memory resident until it is killed or until its size in memory becomes less than or equal to its disk_size (see SIZE_DATASEG).

The calling process associates a logical data segment number (Ldsn) with the data segment. If this Ldsn is already bound to another data segment, 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.

Any error conditions are returned in ErrNum.

```
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 data segment, it can issue a KILL_DATASEG call for that 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.

```
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 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 bound to it. The logical data segment number given must not have a data segment already bound to it. The segment's name is determined by the process which creates the data segment; it cannot be null.

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

When a data segment is opened, it immediately becomes a member of the working set of the calling process. The access mode of the process is Readonly. Use SETACCESS_DATASEG to change the access rights to Readwrite.

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

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

To remove a data segment from the working set of a process, call CLOSE_DATASEG. The data segment referred to by refnum is severed from the context of the calling process, refnum is made invalid, and any reference to the data segment using the original segptr will have unpredictable results. If refnum refers to a local data segment (one created with a null segment name), CLOSE_DATASEG also deletes the data segment. If the data segment is bound to a logical data segment number, CLOSE_DATASEG also frees that LDSN.

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

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

FUNCTION HeapError(hz:THz; BytesNeeded:INTEGER):INTEGER;
BEGIN                            (* handle heap expansion errors *)
WRITELN('Heap is full!  Need ',BytesNeeded,' bytes.');
HeapError:=0;
END;

BEGIN
GrafHeap:='grafheap';
OPEN_DATASEG(ErrorCode,GrafHeap,Heap_Refnum,HeapBuf,1);
IF (ErrorCode=0) THEN            (* grafheap already exists! *)
  BEGIN
  KILL_DATASEG(ErrorCode,GrafHeap);
  CLOSE_DATASEG(ErrorCode,Heap_Refnum);
  END;
MAKE_DATASEG(ErrorCode,GrafHeap,HeapSize,Heap_RefNum,HeapBuf,1);
InitHeap(POINTER(HeapBuf),POINTER(HeapBuf+HeapSize),@HeapError);
END;
```

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

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

FLUSH_DATASEG writes the contents of the data segment identified by refnum to the disk. This call has no effect upon the memory residence or binding of the data segment.

```
SIZE_DATASEG (Var ErrNum:Integer;
              Refnum:Integer;
              deltaMemSize:LongInt;
              Var NewMemSize:LongInt;
              deltaDiskSize:LongInt;
              Var NewDiskSize:LongInt)
```

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

SIZE_DATASEG changes the memory and 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. Because the actual allocation is done in terms of pages (512 byte blocks), the newMemSize and newDiskSize returned by SIZE_DATASEG may be larger than the oldsize plus deltaSize of the respective areas.

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

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

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

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

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

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

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

        ErrNum: Error indicator
        Ldsn:   logical data segment number
        RefNum: data segment identifier
```

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

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

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

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

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

UNBIND_DATASEG(Var ErrNum:Integer;
               RefNum:Integer);

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

BIND_DATASEG binds the data segment referred to by refnum to its associated logical data segment number(s). UNBIND_DATASEG unbinds the data segment from its ldsn's. BIND_DATASEG causes the data segment to become a member of the current working set. At the time of the BIND_DATASEG call, the necessary ldsn's must be available. UNBIND_DATASEG frees the associated ldsn's. A reference to the contents of an unbound segment gives unpredictable results. OPEN_DATASEG and MAKE_DATASEG determine which ldsn's are associated with a given data segment.

CHAPTER 4

EXCEPTIONS AND EVENTS

EXCEPTIONS and EVENTS

Processes have several ways to keep informed about the state of the world. Normal-process-to process communication and synchronization can be handled using events or shared data segments. An abnormal condition can cause an exception (interrupt) to be signalled which the process can respond to in whatever way it sees fit.

EXCEPTIONS

Normal execution of a process can be interrupted by an exceptional condition (such as division by zero or address error). Some of these conditions are trapped by the hardware, some by the system software, and others can be signalled by the process itself. Exceptions have character string names, some of which are predefined and reserved by the Operating System.

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

* Enabled

* Queued

* Ignored

If the exception is enabled, the system next looks for a user defined handler for that exception. If none is found, the system default exception handler is invoked. It usually aborts the current process.

If the state of the exception is queued, the exception is placed on a queue. When that exception is subsequently enabled, this queue is examined, and if any exceptions are found, the appropriate exception handler is entered. Processes can flush the exception queue.

If the state of the exception is ignored, the system still detects the occurrence of the exception, but the exception is neither honored nor queued.

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

SYSTEM DEFINED EXCEPTIONS

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

* Division by zero (SYS_ZERO_DIV). Default handler aborts process.

* Value out of bounds (SYS_VALUE_OOB). Default handler aborts process.

* Overflow (SYS_OVERFLOW). Default handler aborts process.

* Process termination (SYS_TERMINATE). This exception is signalled when a process terminates, or when there is a bus error, address error, illegal instruction, privilege violation, or line 1010 or 1111 emulator error. The default handler does nothing.

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.

EXCEPTION HANDLERS

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

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

If an exception handler for a given exception already exists when another handler is declared for that exception, the old one becomes disassociated. There is no notion of block structured declaration of exception handlers.

An exception can occur during the execution of an exception handler. The state of the exception determines whether it is queued, honored, 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.

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

The following code fragment gives an example of exception handling.

```
PROCEDURE Handler(Env_Ptr:p_env_blk;
                  Data_Ptr:p_ex_data);
VAR ErrNum:INTEGER;
BEGIN
(* Env_Ptr points to a record containing the program counter, *)
(* and all registers.  Data_Ptr points to an array of 12 longints *)
(* that contain the event header and text if this handler is *)
(* associated with an event-call channel (see below) *)
.
.
.
ENABLE_EXCEP(errnum,excep_name);
.
.
.
END;
.
(* this is either in a different segment or at the top level *)
.
Excep_name:='EndOfDoc';
DECLARE_EXCEP_HDL(errnum,excep_name,@Handler);
.
.
.
SIGNAL_EXCEP(errnum,excep_name,excep_data);
.
.
.
```

At the time the exception handler is invoked, the stack is:

```
   low             |                        |
 address           |                        |        Exception Handler
                   |                        |
                   +------------------------+
                   |         Link           |
                   +------------------------+
                   | @Return_Exception      |
                   +------------------------+
                   |     Data_Ptr           |----+
                   +------------------------+    |
              +----|     Environment_Ptr    |    |
              |    +------------------------+    |
              |    |     Terminate Flag     |    |
              |    +------------------------+    |
              |    | Exception Kind         |<---+
              |    | Function Code (fc)     |        Exception Data Block
              |    | Access Address (aa)    |       (Sys_Terminate exception)
              |    | Instruction Register   |
              |    | Status Register        |
              |    | Program Counter        |
              |    +------------------------+
              +--->| Program Counter        |        Exception Environment Block
                   | Status Register        |
                   | D0--D7 and A0--A7      |
                   +------------------------+
                   |         Link           |
                   +------------------------+
                   | Program Counter        |
                   +------------------------+
   high            |                        |
 address           |                        |
                   |                        |
```

The Exception Data Block given here reflects the state of the stack upon a SYS_TERMINATE exception. The term_ex_data record described in the Interface appendix gives the various forms the data block can take. The status register and program counter values in the data block reflect the true (current) state of these values. The same data in the Environment block reflects the state of these values at the time the exception was signalled, not the values at the time the exception actually occurs.

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

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

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.

EVENT CHANNELS

Event channels can be viewed as a higher-level approach to pipes. The most important difference is that event channels deal with fixed size data blocks, whereas pipes can handle an arbitrary byte stream.

An event channel can be globally or locally defined. A global event channel has a globally defined pathname catalogued in the file system, and can be used by any process to handle user defined events. A local event channel, however, has no name and is known only by the Operating System and the process that opened it.

A local event channel is automatically created when a process is created. This channel can be opened by the father process to receive system generated events pertaining to its son.

There are two types of 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, treats its event as an exception. The exception name must be given when the event-call event channel is opened, and an exception handler for that exception must be declared. When an event is sent to an event-call event channel, the Operating System signals the associated exception. If the process reading the event-call channel is suspended at the time the event is sent, the event is queued and is executed 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.

The following code fragment uses event-wait channels to handle process synchronization:

```
PROCESS A                           PROCESS B
---------                           ---------

Open Chn_1 to receive;              Open Chn_1 to send;

Open Chn_2 to send;                 Open Chn_2 to receive;

REPEAT                              REPEAT

    Send to Chn_2;                      Wait for Chn_2;

    Wait for Chn_1;                     Send to Chn_1;

UNTIL AllDone;                      UNTIL AllDone;
```

The order of execution of the two processes is the same regardless of the process priorities. In the following example using event-call channels, however, the process priorities do affect the order of execution.

```
PROCESS A                           PROCESS B
---------                           ---------

Declare Excep_1;                    Declare Excep_2;

Open Chn_1 to receive Excep_1;      Open Chn_1 to send;

Open Chn_2 to send;                 Open Chn_2 to receive Excep_1;

Send Chn_2;

PROCEDURE Handler;                  PROCEDURE Handler;
   Send Chn_2;                          Send Chn_1;
   Yield_Cpu;                           Yield_Cpu;
```

THE SYSTEM CLOCK

A process can read the system clock time, convert to local time, or delay its own continuation until a given time. The year, month, day, hour, minute, second, and millisecond are available from the clock. The system clock is in Greenwich mean time.

EXCEPTION MANAGEMENT CALLS

The event and exception management routines use several special types and constants. To save space and reduce redundancy, these types are defined only in Appendix A, and are referred to in the rest of this chapter without much further comment.

```
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 informs the Operating System that the occurrence of the exception referred to by excep_name should cause the execution of the exception handler whose address is given by entry_point. Excep_name is a character string name that is locally defined in the process and known only to the process and the Operating System. If entry-point is nil, the system default exception handler for that exception is used. Any previously declared exception handler is disassociated by this call. The exception itself is automatically enabled.

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

If DECLARE_EXCEP_HDL is called with an exception handler address of @NIL and there is no system default handler for the exception, the exception will have no handler defined.

```
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:

1) 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.

2) 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 also succeeds.

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

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

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

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

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

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

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

Once Ex_occurred_f has been set to true, it is reset to false only by a call to FLUSH_EXCEP.

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

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

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

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

```
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.

EVENT MANAGEMENT CALLS

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

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

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

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

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

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

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

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

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

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

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

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

```
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.

```
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 information includes:

```
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 this channel *)
       Open_send:integer;     (* no. of processes sending to this channel *)
       Ec_name:pathname;      (* exception name for event-call *)
       END;
```

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

ErrNum:	Error indicator
Wait_list:	Record with array of event channels
Refnum:	Identifier of channel containing 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 pointing to the list of event channel identifiers. If, on the other hand, it must wait for all the events from a set of channels, an "and" condition, then for each channel in the set, WAIT_EVENT_CHN should be called with a wait_list pointing just to that channel.

The structure of t_waitlist is:

```
Record
    Length:Integer;
    Refnum:Array[0..size_waitlist] of Integer;
    End;
```

P_r_eventblk is a pointer to a record containing the event header and the event text.

Currently the possible event type values are:

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

If you call WAIT_EVENT_CHN on an event-call channel which 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 which does not have any queued events, an error is returned.

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

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

FLUSH_EVENT_CHN clears out the specified event channel. All events queued in the channel are removed.

```
SEND_EVENT_CHN (Var ErrNum:Integer;
                Refnum:Integer;
                Event_ptr:p_s_eventblk;
                Interval:t_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 contains only the event text; the header is added by the system.

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

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

The parameter, interval, indicates whether the event is a timed event. T_interval is a record containing a day and a millisecond field. If both fields are 0, the event is sent immediately. If the day 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 day field is greater than or equal to 0, the event is sent that number of days and milliseconds from the present. The time given in time_rec is in Greenwich Mean Time.

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.

CLOCK CALLS

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

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

DELAY_TIME stops execution of the calling process for the number of days and milliseconds specified in the interval record. If this time period is zero, DELAY_TIME obviously has no effect. If the period is less than zero, execution of the process is delayed until the time specified by Clktime in Greenwich Mean Time. Time_rec is a record defined as:

```
        time_rec = RECORD
                        Year:Integer;
                        Day:1..366;
                        Hour:-23..23;
                        Minute:-59..59;
                        Second:0..59;
                        Msec:0..999;
                   END;
```

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

        ErrNum:   Error indicator
        GMT_Time: Time information
```

GET_TIME returns the current system clock time in the record GMT_Time.

```
        time_rec = RECORD
                       Year:Integer;
                       Day:1..366;
                       Hour:-23..23;
                       Minute:-59..59;
                       Second:0..59;
                       Msec:0..999;
                   END;
```

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

ErrNum: Error indicator
Hour: Number of hours difference from Greenwich Mean Time
Minute: Number of minutes difference from Greenwich Mean Time

SET_LOCAL_TIME_DIFF informs the Operating System of the difference in hours and minutes between the local time and Greenwich Mean Time (that is, GMT-localTime). Hour and Minute can be negative.

```
CONVERT_TIME (Var ErrNum:Integer;
              Var GMT_Time:Time_rec;
              Var Local_Time:Time_rec;
              To_gmt:Boolean)
```

ErrNum:	Error indicator
GMT_Time:	Greenwich Mean Time
Local_Time:	Local time
To_gmt:	Direction of time conversion

CONVERT_TIME converts between local time and system clock time. The system clock is in Greenwich Mean Time. To_gmt is a boolean value indicating which direction the conversion is to go. If it is true, the system takes the time data in local_time and puts the corresponding GMT time in gmt_Time. Otherwise, it takes the time data in gmt_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 5

SYSTEM CONFIGURATION AND STARTUP

SYSTEM CONFIGURATION AND STARTUP

SYSTEM STARTUP

Startup is a multi-step operation. After the startup request is generated, code in the bootstrap ROM executes. This code runs a series of diagnostic tests, and signals by a beep that all is well.

The ROM next selects a boot device. The default boot device is the Twiggy drive 1, but this can be overridden by the keyboard or by parameter memory. The ROM passes the memory size, the boot device position, and the results of the diagnostics to the loader found on the boot device.

The loader allocates physical memory and loads three types of Operating System segments needed during Startup, including the configurable device drivers. It creates a pseudo-outer-process, enters the Operating System, and passes to Startup a physical address map and some parameter data.

Startup inherits the unmapped address space of the loader, initializes the memory map, initializes all the Operating System subsytems, creates the system process, then destroys the pseudo-outer-process (itself), passing control to the highest priority process. At this point the boot process is complete and the outer shell process or the Filer is in control.

SELF-DIAGNOSTICS

The self-test code in ROM performs an overall diagnostic check at power-up and then executes the bootstrap routine from the disk.

The first tests initialize various system controls; MMU registers, contrast control, parity logic, etc. You should hear a beep notifying you that the startup tests have begun. A checksum is done on the ROM itself, then all of the RAM in the system is tested for shorts and address uniqueness. The Memory Management Unit is also tested in this manner.

Parts of the video and parity generator/checker circuitry are tested next. The keyboard and mouse interfaces are tested by checking various modes of the Versatile Interface Adapter operation, and by running a ROM/RAM test of all the processors used in the interfaces. Meanwhile, the disk controller is running its own tests of ROM and RAM. Finally, the RS232 port and the clock are tested.

CUSTOMIZING YOUR SYSTEM

The features and design of the system configuration program have not yet been defined.

APPENDICES

OPERATING SYSTEM INTERFACE

```
CONST
    Max_ename = 32;          (* max length of file system object name *)
    Len_exname = 16;         (* exception name length *)
    Size_exdata = 11;        (* 48 bytes in exception data block *)
    Size_etext = 9           (* 40 bytes of event text *)
    Size_waitlist = 10;      (* current size of wait 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 killed by another process *)
    fthr_term = 4;           (* process's father is terminating *)

    def_div_zero = 11;       (* default handler called for SYS_ZERO_DIV *)
    def_value_oob = 12;      (* default handler called for SYS_VALUE_OOB *)
    def_ovfw = 13;           (* default handler called for SYS_OVERFLOW *)
    def_nmi_key = 14;        (* default handler called for NMI key excep *)
    def_range = 15;          (* SYS_VALUE_OOB due to value range error *)
    def_str_index = 16;      (* SYS_VALUE_OOB due to string index error *)

    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 *)

    div_zero = 31;           (* hardware exception kind definitions *)
    value_oob = 32;
    ovfw = 33;
    nmi_key = 34;
    value_range = 35;
    str_index = 36;


TYPE

Pathname = STRING[255];
E_Name = STRING[Max_Ename];
NameString = STRING[20];
Accesses = (DRead, DWrite, Append, Private, Global_Access);
MSet = SET OF Accesses;
IoMode = (Absolute, Relative, Sequential);
Uid = INTEGER;
Info_Type = (device_t, volume_t, object_t);
Devtype = (diskdev, pascalbd, seqdev, bitbkt, non_io);
Filetype = (undefined, MDDFFile, rootcat, freelist, badblocks,
            sysdata, spool, exec, usercat, pipe, bootfile,
            swapdata, swapcode, ramap, userfile, killedobject);
Entrytype = (emptyentry, catentry, linkentry, fileentry, pipeentry,
             ecentry, killedentry);
```

```
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;
                              freecount: LONGINT;
                              DTVC,
                              DTVB,
                              DTVS:       LONGINT;
                              Machine_id,
                              overmount_stamp,
                              master_copy_id: LONGINT;
                              privileged,
                              write_protected: BOOLEAN;
                              master,
                              copy,
                              scavenge_flag: BOOLEAN);
                    object_t:(size:      LONGINT;
                              psize:     LONGINT; (*physical size in bytes*)
                              lpsize:    INTEGER; (*Logical page size in bytes*)
                              ftype:     filetype;
                              etype:     entrytype;
                              DTC,
                              DTA,
                              DTM,
                              DTB:       LONGINT;
                              refnum:    INTEGER;
                              fmark:     LONGINT;
                              acmode:    mset;
                              nreaders,
                              nwriters,
                              nusers:    INTEGER;
```

```
                              fuid:     uid;
                              eof,
                              safety_on,
                              kswitch:  BOOLEAN;
                              private,
                              locked,
                              protected:BOOLEAN);
          END;


ProcInfoRec = RECORD
                ProgPathName:Pathname;
                Global_Id    :LONGINT;
                Priority     :1..255;
                State        :(Pactive,PSuspended,Pwaiting);
                Data_In      :Boolean
              END;

DsInfoRec =
    RECORD
        Mem_Size:LONGINT;
        Disc_size:LONGINT;
        NumbOpen:INTEGER;
        Ldsn:INTEGER;
        BoundF:BOOLEAN;
        PresentF:BOOLEAN;
        CreatorF:BOOLEAN;
        RWAccess:BOOLEAN;
        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 block *)
t_ex_sts = RECORD             (* exception status *)
             Ex_occurred_f:BOOLEAN;
             ex_state:t_ex_state;
             num_excep:INTEGER; (* no. of exceptions queued *)
             Hdl_adr:Longadr;
           END;

P_env_blk = ^env_blk;
Env_blk = RECORD              (* environment block for handler *)
             PC:LONGINT;        (* program counter *)
             SR:INTEGER;        (* status register *)
             D0,D1,D2,D3,D4,D5,D6,D7:LONGINT;
             A0,A1,A2,A3,A4,A5,A6,A7:LONGINT
          END;
```

```
p_term_ex_data = ^term_ex_data;
term_ex_data = RECORD           (* SYS_TERMINATE exception data block *)
                CASE execp_kind:LONGINT OF
                  call_term,
                  ended,
                  self_killed,
                  killed,
                  fthr_term:(); (* due to process termination *)
                  illg_inst,
                  priv_violation,
                  line_1010,
                  line_1111,
                  def_div_zero,
                  def_value_oob,
                  def_ovfw,
                  def_nmi_key:
                            (SR:INTEGER;
                             PC:LONGINT);
                  def_range,
                  def_str_index:(value_check:INTEGER;
                                 upper_bound:INTEGER;
                                 lower_bound:INTEGER;
                                 return_pc:LONGINT;
                                 caller_a6:LONGINT);
                  bus_error,
                  addr_error:
                    (fun_field:PACKED RECORD    (* one INTEGER *)
                               filler:0..$7FF; (* 11 bits *)
                               r_w_flag:BOOLEAN;
                               i_n_flag:BOOLEAN;
                               fun_code:0..7;
                             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
                  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;
```

```
T_waitlist = RECORD
                 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;
              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;

t_interval = RECORD
                 Day:INTEGER;       (* number of days *)
                 Millisec:LONGINT;(* number of millisecond in day *)
                                    (* should be 0..86399999 *)
               END;

time_rec = RECORD
                 Year:INTEGER;
                 Day:1..366;
                 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;
                 Num_events:INTEGER;
                 Open_recv:INTEGER;
                 Open_send:INTEGER;
                 Ec_name:pathname;
              END;

Hour_range = -23..23;
Minute_range = -59..59;
```

```
(* 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 KILL_OBJECT
            (VAR  Ecode:INTEGER;
             VAR  Path:Pathname)

PROCEDURE RENAME_ENTRY
             (VAR  Ecode:INTEGER;
              VAR  Path:Pathname;
              VAR  Newname:E_name)

PROCEDURE LOOKUP
        (VAR  Ecode:INTEGER;
         VAR  Path:Pathname;
         Index:INTEGER;
         VAR  Attributes:Fs_Info)

PROCEDURE INFO
      (VAR  Ecode:INTEGER;
       Refnum:INTEGER;
       VAR  RefInfo:Fs_Info)

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 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 DEVICE_CONTROL
             (VAR  Ecode:INTEGER;
              VAR  Path:Pathname;
              Ccode, CParm:INTEGER)

PROCEDURE ALLOCATE
        (VAR  Ecode:INTEGER;
         Refnum:INTEGER;
         Contiguous:BOOLEAN;
         Count:LONGINT;
         VAR  Actual:LONGINT)

PROCEDURE COMPACT
        (VAR  Ecode:INTEGER;
         Refnum:INTEGER)

PROCEDURE TRUNCATE
        (VAR  Ecode:INTEGER;
         Refnum:INTEGER)

PROCEDURE FLUSH
      (VAR  Ecode:INTEGER;
       Refnum:INTEGER)

PROCEDURE SET_SAFETY
          (VAR  Ecode:INTEGER;
           VAR  Path:Pathname;
           On_off:BOOLEAN)

PROCEDURE SET_WORKING_DIR
             (VAR  Ecode:INTEGER;
```

```
                VAR   Path:Pathname)

PROCEDURE GET_WORKING_DIR
               (VAR   Ecode:INTEGER;
                VAR   Path:Pathname)

PROCEDURE MOUNT
        (VAR   Ecode:INTEGER;
         VAR   VName:E_name;
         VAR   Password, Devname:E_name)

PROCEDURE UNMOUNT
        (VAR   Ecode:INTEGER;
         VAR   VName:E_name)

PROCEDURE RESET_CATALOG
               (VAR ecode:INTEGER;
                VAR Path:Pathname)

PROCEDURE Get_NEXT_ENTRY
               (VAR Ecode:INTEGER;
                VAR Prefix,Entry:E_Name)

(* Process Management System Calls *)

PROCEDURE MAKE_PROCESS
             (VAR   ErrNum:INTEGER;
              VAR   Proc_Id:LONGINT;
              VAR   ProgFile:Pathname;
              VAR   EntryName:NameString;
              Evnt_chn_refnum:INTEGER)

PROCEDURE TERMINATE_PROCESS
                (VAR   ErrNum:INTEGER;
                 Event_ptr:P_S_Eventblk)

PROCEDURE INFO_PROCESS
             (VAR   ErrNum:INTEGER;
              Proc_Id:LONGINT;
              VAR   Proc_Info:ProcInfoRec)

PROCEDURE KILL_PROCESS
             (VAR   ErrNum:INTEGER;
              Proc_Id:LONGINT)

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 SETPRIORITY_PROCESS
                (VAR  ErrNum:INTEGER;
                 Proc_Id:LONGINT;
                 New_Priority:INTEGER)

PROCEDURE YIELD_CPU
             (VAR Errnum:INTEGER;
              To_Any:BOOLEAN)

FUNCTION MY_ID:LONGINT

(* Memory Management System Calls *)

PROCEDURE MAKE_DATASEG
             (VAR  ErrNum:INTEGER;
              VAR  SegName:Pathname;
              Mem_Size,Disk_Size:LONGINT;
              VAR  RefNum:INTEGER;
              VAR  SegPtr:LONGINT;
              Ldsn:INTEGER)

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 FLUSH_DATASEG
              (VAR  ErrNum;
               RefNum:INTEGER)

PROCEDURE SIZE_DATASEG
             (VAR  ErrNum:INTEGER;
              RefNum:INTEGER;
              DeltaMemsize:LONGINT;
              VAR  NewMemSize:LONGINT;
              DeltaDiskSize:LONGINT;
              VAR NewDiskSize:LONGINT)

PROCEDURE INFO_DATASEG
             (VAR  ErrNum:INTEGER;
              RefNum:INTEGER;
              VAR  DsInfo:DsInfoRec)
```

```
PROCEDURE SETACCESS_DATASEG
                 (VAR  ErrNum:INTEGER;
                  RefNum:INTEGER;
                  Readonly:BOOLEAN)

PROCEDURE BIND_DATASEG
          (VAR  ErrNum:INTEGER;
           Ldsn:INTEGER)

PROCEDURE UNBIND_DATASEG
          (VAR  ErrNum:INTEGER;
           RefNum:INTEGER)

PROCEDURE INFO_LDSN
          (VAR  ErrNum:INTEGER;
           Ldsn:INTEGER;
           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 INFO_EXCEP
               (VAR  ErrNum:INTEGER;
                VAR  Excep_Name:t_ex_name;
                VAR  Excep_status:t_ex_sts)

PROCEDURE SIGNAL_EXCEP
               (VAR  ErrNum:INTEGER;
                VAR  Excep_Name:t_ex_name;
                VAR  Excep_data: t_ex_data)

PROCEDURE FLUSH_EXCEP
               (VAR  ErrNum:INTEGER;
                VAR  Excep_Name:t_ex_name)

(* Event 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:t_interval;
               Clktime:Time_rec)
```

```
(* Timer Function System Calls *)

PROCEDURE DELAY_TIME
               (VAR  ErrNum:INTEGER;
                Interval:T_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)
```

System Reserved Exception Names

SYS_OVERFLOW	overflow exception. Signalled if the TRAPV instruction is executed, and the overflow condition is on.
SYS_VALUE_OOB	value out of bound exception. Signalled if the CHK instruction is executed, and the value is less than 0 or greater than upper bound.
SYS_ZERO_DIV	division by zero exception. Signalled if the DIVS or DIVU instruction is executed, and the divisor is zero.
SYS_TERMINATE	termination exception. Signalled when a process is to be terminated.
SYS_SHUT_OFF	system shut off exception. When the system is to be shut off, this exception is signalled to every process to save the current state.
SYS_POWER_ON	system power on exception. After the system is powered on, this exception is signalled to every process to continue where it left off when system was shut off.

System Reserved Event Types

SYS_SON_TERM	"son terminate" event type. This event is sent to the father process when a son process makes a TERMINATE_PROCESS call.

ERROR CODES

0	no error
1	invalid refnum
5	parity error

PROCESS MANAGEMENT

100	Specified process does not exist
101	Specified process is a system process
110	invalid priority specified (must be 1..255)
115	specified process is already suspended (Suspend_process)
120	specified process is already active (Activate_Process)
125	sepcified process is already terminating (Kill_Process)
130	can not open program file
131	error while trying to read program file
132	invalid program file (not executable)
133	cannot make process stack for new process
134	cannot make process syslocal for new process
135	cannot get a PCB for the new process
136	cannot set up communication channel for new process
137	program uses an invalid intrinsic unit (either names do not agree, or unit is not intrinsic)
138	cannot access program file during loading
139	cannot get a PLCB (program load control block) for program--out of sysglobal space
140	program uses an invalid shared segment (either names do not agree, or segment is not in Intrinsic.Lib)
141	cannot access a shared library file while loading

EXCEPTION MANAGEMENT

201	no such exception name declared
202	no space left in the system data area for declare_execp_hdl or signal_excep.

MEMORY MANAGEMENT

301	input refnum is invalid
302	input ldsn value is invalid
303	no data segment bound to an ldsn when there should be
304	data segment bound to an ldsn when it shouldn't be
305	data segment already bound to an 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 memory size <= 0 memory size of shared data segment > 128K disk size < 0

EVENT MANAGEMENT

401	invalid event channel name passed to make_event_chn: empty string or string longer than 16 characters
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 to make_event_chn, kill_event_chn, or open_event_chn
410	attempt to open a local event channel to send
411	attempt to open an event channel to receive when event channel already has a receiver
412	calling process has already opened this channel to send or receive
414	attempt to open channel that is being killed
415	warning: wrong number of bytes in channel when open
420	attempt to wait on a channel that the calling process did not open
421	wait_event_chn returns while waiting on an empty channel because a sender process was not able to successfully complete sending an event.
422	attempt to call wait_event_chn on an empty event-call channel
423	cannot find corresponding event channel after being blocked (wait_event_chn)
424	the actual amount of data returned while reading an event from a channel is not the same as the size of an event block in wait_event_chn (probably disk I/O failure)
425	event channel empty after being unblocked (wait_event_chn)
430	attempt to send to a channel which the calling process does not have open
431	the actual amount of data transferred while writing an event to a channel is not the same as the size of an event block in send_event_chn (disk is probably full)
440	warning: wrong number of bytes in channel when Info_Event_Chn called

TWIGGY DISK ERRORS

611	unexpected interrupt from drive 2
612	unexpected interrupt from drive 1
613	illegal disk address or transfer length
614	no disk present in drive

TIME MANAGEMENT

630	the time passed to delay_time, convert_time, or send_event_chn is such that the year is less than 1890 or greater than 2069.
635	process got unblocked prematurely due to process termination (delay_time)

636	timer request did not complete successfully in delay_time
638	the time passed to delay_time or send_event_chn is more than 230 days from the current GMT time

RS-232

640	RS-232 driver called with wrong version number
641	RS-232 read or write initiated with illegal parameter
643	Unexpected RS-232 interrupt
644	Illegal refnum used to call T_DISABLE from within RS-232 driver
645	Illegal refnum used to call T_RE_ENABLE from within RS-232 driver
646	No memory available to initialize RS-232
647	Unexpected RS-232 timer interrupt
648	Attempt to send unpermitted command to serial controller card

STARTUP

700	Mismatch between loader version number (in OS.OBJ) and operating system version number (in SYSTEM.OS.OBJ)
701	OS exhausted its internal space during startup
702	Cannot make system process
703	Cannot kill pseudo-outer process
704	Cannot create driver
705	Cannot program NMI key
706	Cannot (soft) initialize Twiggy
707	Cannot (soft) initialize the file system volume
708	Profile not readable

FILE SYSTEM

VmStuff:

801	IoResult <> 0 on I/O using the Monitor (LISAIO)
802	Asynchronous I/O request not completed successfully
806	Page specified is out of range (TFDM)
809	Invalid arguments (page, address, offset, or count) (VM)
816	Not enough sysglobal space for file system buffers (initqvm)
819	Bad device number (IO_INIT)
820	No space in sysglobal for asynchronous request list
821	Already initialized I/O for this device
822	Bad device number (IO_DISINIT)

SFileIO:

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	s-file number < 0 or > maxfiles (illegal value) (SList_IO)
837	Unallocated s-file or I/O error (FMap_Mgr)
838	Map overflow: s-file too large
841	Unallocated s-file or I/O error (Get_PSize)
843	Requested exact fit, but one couldn't be provided (AppendPages)
847	Requested transfer count is <= 0 (DataIO)
848	End-of-file encountered
849	Invalid page or offset value in parameter list
852	Bad unit number (FlushFS)

854	No free slots in s-list directory (too many s-files) (New_SFile)
855	No available disk space for file hints
856	Device not mounted
857	Empty, locked, or invalid s-file (Kill_SFile)
861	Relative page is beyond PEOF (bad parameter value) (AbsPage)
864	No sysglobal space for volume bitmap (Real_Mount, Real_Unmount)
866	Wrong FS version or not a valid Lisa FS volume
867	Bad unit number (Real_Mount, Real_Unmount)
868	Bad unit number (Def_Mount, Def_Unmount)
869	Unit already mounted (mount)/no unit mounted (unmount)
870	No sysglobal space for DCB or MDDF (mount)
FS_Primitives:	
871	Parameter not a valid s-file ID (Open_SFile)
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 (Close_SFile)
879	Attempt to postion past LEOF (Direct_IO)
881	Attempt to read empty file (FileIO)
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 nor update filesize hints after a write (fileio)
887	Catalog pointer does not indicate a catalog (bad parameter)
888	Entry not found in catalog (Lookup_by_ename)
890	Entry by that name already exists (Make_Entry)
891	Catalog is full, or was not as catalog
892	Illegal name for an entry
894	Entry not found, or not a catalog (Kill_Entry)
895	Invalid entry name (kill_entry)
896	Safety switch is on--cannot kill entry (kill_entry)
FS_Init:	
897	Invalid bootdev value
FS_Interface:	
921	Pathname invalid or no such device (Make_File)
922	Invalid label size (Make_File)
926	Pathname invalid or no such device (Make_Pipe)
927	Invalid label size (Make_Pipe)
941	Pathname invalid or no such device (Kill_Object)
946	Pathname invalid or no such device (Open)
947	Not enough space in syslocal for file system refdb
948	Entry not found in specified catalog (Open)
949	Private access not allowed if file already open shared
950	Pipe already in use, requested access not possible
951	File is already opened in private mode (open)
952	Bad refnum (Close_Object)
954	Bad refnum (Read_data)
955	Read access not allowed to specified object
956	Attempt to position FMARK past LEOF not allowed
957	Negative request count is illegal (read_data)
958	Non-sequential access is not allowed (read_data)
959	System resources exhausted
960	Error writing to pipe while an unsatisfied read was pending
961	Bad refnum (write_data)

962	No WRITE or APPEND access allowed
963	Attempt to position FMARK too far past LEOF
964	Append access not allowed in absolute mode
965	Append access not allowed in relative mode
966	Internal inconsistency of FMARK and LEOF (warning)
967	Non-sequential access is not allowed (write_data)
968	Bad refnum (Flush)
971	Pathname invalid or no such device (Lookup)
972	Entry not found in specified catalog
974	Bad refnum (Info)
977	Bad refnum (allocate)
978	Page count is non-positive (allocate)
979	Not a block structured device (allocate)
981	Bad refnum (Truncate)
982	No space has been allocated for specified file
983	Not a block structured device (truncate)
985	Bad refnum (Compact)
986	No space has been allocated for specified file
987	Not a block structured device (compact)
988	Bad refnum (Flush_Pipe)
989	Caller is not a reader of the pipe
990	Not a block structured device (flush_pipe)
999	Asynchronous read was unblocked before it was satisfied. This may occur during process termination.
1021	Pathname invalid or no such entry (Rename_Entry)
1022	No such entry found (rename_entry)
1023	Invalid newname, check for '-' in string (rename_entry)
1031	Pathname invalid or no such entry (Read_Label)
1032	Invalid transfer count (read_label)
1033	Nop such entry found (read_label)
1041	Pathname invalid or no such entry (Write_Label)
1042	Invalid transfer count (write_label)
1043	No such entry found (write_label)
1051	No device or volume by that name (mount)
1052	A volume is already mounted on device
1061	No device or volume by that name (Unmount)
1062	No volume is mounted on device
1071	Not a valid or mounted volume for working directory
1091	Pathname invalid or no such entry (Set_Safety)
1092	No such entry found (set_safety)
1121	Invalid device, not mounted, or not a catalog (reset_catalog)
1128	Invalid pathname, device, or volume not mounted (get_dev_name)
1196	Something is still open on disk--cannot unmount (real_unmount)
1197	Volume is not formatted or cannot be read (def_mount)
1198	Negative request count is illegal (write_data)
1199	Function or procedure is not yet implemented

The pathname error codes (921, 926, 941, 946, and 971) often mean that the volume specified in the pathname is not mounted. If error 966 occurs while writing a file using the FTP utility, you probably ran out of space on the destination volume.

OS LOADER DIAGNOSTICS

Error Message	Cause or Description
FILE SYSTEM VERSION MISMATCH	When booting from the Twiggy
FILE SYSTEM CORRUPT	When booting from the Twiggy
MEMORY EXHAUST	You forgot to run SETSP, or used an incorrect value
SYSTEM CODE FILE NOT FOUND	Cannot find SYSTEM.OS.OBJ
SYSTEM CONFIGURATION FILE NOT FOUND	Nor does it exist yet
BOOT DEVICE READ FAILED	IoResult was not 0 for whatever reason while trying to read SYSTEM.OS.OBJ
PROGRAM NOT EXECUTABLE	Refers to SYSTEM.OS.OBJ
CODE FILE CORRUPT	Refers to SYSTEM.OS.OBJ
TOO MANY OS SEGMENTS	
UNKNOWN BOOT ERROR	