Mesa System Documentation

Version 5.0
April 1979

This document describes configurations of the Alto/Mesa system software and the individual components which comprise them. These components include runtime support for the language, routines for manipulating the Alto display, keyboard, and file system, facilities for loading client programs and building new systems, and a number of useful common software packages.

XEROX
SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

Table of Contents

Preface

April 1979

This document describes configurations of the Alto/Mesa system software and the individual components which comprise them. These components include runtime support for the language, routines for manipulating the Alto display, keyboard, and file system, facilities for loading client programs and building new systems, and a number of useful common software packages.

Suggestions as to the form, correctness, and understandability of this material should be funneled through your support group. All of us involved in the development of Mesa welcome feedback and suggestions on both the language and the system and debugging environment.

1. Overview

The Alto/Mesa environment contains all of the facilities described in this document. **Section 2** contains an enumeration of the definitions modules available in the Alto/Mesa system; it includes a brief description of the facilities provided by each interface. Complete documentation can be found in **Section 5.**

Section 3 describes the various configurations and optional packages that comprise the Alto/Mesa system. Two configurations of the system are available: the standard one, which includes most of the facilities described here, and a more basic one, containing only required runtime and file system support (it contains no display or keyboard software, for example). A number of optional packages are available for use with the basic system, so that the Mesa system can be tailored to each application.

The *Mesa Executive* **(Section 4)** is a simple (and small) user interface supporting only a few commands. It allows only program loading, state saving, and debugger communication; more complicated operations must be provided by the user or by one of the standard packages that is available separately. The *Mesa Executive* also supports command line input in both the standard and the basic system; this allows the user interface to be completely controlled by the application.

Finally, the bulk of this manual **(Section 5)** contains detailed descriptions of each of the packages summarized in **Section 2.**

Further details are available elsewhere on the language (*Mesa Language Manual*) and the debugger (*Mesa Debugger Documentation*). Information on obtaining and running the system is contained in the *Mesa User's Handbook.*

2. Definitions and Interfaces

The following list enumerates the public system modules of interest to Alto/Mesa programmers. The interface name is followed by the title of the section of this document (or other reference material) which describes the facilities the interface provides. The parenthesized names following the DEFINITIONS module identify the programs which implement that interface -- the ultimate documentation after all.

AllocDefs (Swapper) Segment Package

Low level memory allocation functions are defined here. Complete control of segment allocation and swapping can be obtained using this interface.

AltoDefs (Hardware) *Mesa Language Manual*

Contains a number of machine dependent constants describing physical characteristics of the Alto and its basic data types (bits per character, characters per word, etc.).

AltoDisplay (Hardware) *Alto: A Personal Computer System; Hardware Manual*

Provides a Mesa definition of the properties of the Alto display: its dimensions, **Resolution**, **Background**, the **DCB** chain, and the **DCB** format. A **LongDCB** has been added for interim use on the D0.

AltoFileDefs (BCPL) *Alto Operating System Reference Manual*

Defines the data structures (but not the operations) used in manipulating the Alto file system. Note that these structures are shared by several software systems running on the Alto.

BitBltDefs (Hardware) *Alto: A Personal Computer System; Hardware Manual*

Provides a Mesa definition of the hardware BitBlt (bit boundary block transfer) operation. A **LongBBTable** has been added for interim use on the D0.

DirectoryDefs (Directory) Directory Package

Common operations on the Alto's file directory are defined here; they are all based on a single primitive which enumerates entries in a directory.

DisplayDefs (SystemDisplay, DisplayControl) Display Package

Provides device-dependent operations for the display. Simple operations are implemented using the standard stream interface (see **StreamDefs**).

DoubleDefs (Double) Miscellaneous

Contains a complete set of procedures for operating on long (double precision) CARDINALS (see also **InlineDefs**). This package is no longer supported, since LONG CARDINALS are fully supported by the language.

FontDefs (AlFont) Display Package

Defines a uniform interface for all font formats. It includes facilities for dynamically swapping font files.

FrameDefs (Miscellaneous, Modules, Swapper, UnNewConfig) Modules

Provides low level operations on modules and their runtime representation (global frames). It includes facilities for controlling residency of a module's code segment and for loading and unloading configurations.

FSPDefs (FSP) Storage Management

This memory allocation package provides temporary storage for small, transient data structures whose size is not known at compile time.

ImageDefs (CheckPoint, MakeImage) Image Files

Image files are used to save the state of a computation so that it can be restarted later (perhaps in a different environment). Different types of image files can be created using the procedures defined in this interface; each type makes different assumptions about the state of the environment when it is restarted.

InlineDefs (InlineDefs) Miscellaneous

Defines a set of instructions not accessible at the language level. Logical operations and some extended precision arithmetic is included.

IODefs (StreamIO) StreamIO Package

A simple Teletype style I/O interface is provided by these procedures. Minimal editing and input and output conversion routines are included.

KeyDefs (Keyboard, KeyStreams) Keyboard Package

Provides device-dependent operations for the keyboard, keyset, and mouse. Simple operations are implemented using the standard stream interface (see **StreamDefs**).

MiscDefs (Miscellaneous) Miscellaneous

A set of miscellaneous but useful procedures that don't obviously belong in any of the other interfaces.

Mopcodes (Hardware) *OIS Processor Principles of Operation*

Defines the opcode numbers used by the Mesa instruction set. It is useful when inline procedures need to be defined to obtain access to machine facilities not available in the language.

OsStaticDefs (BCPL) *Alto Operating System Reference Manual*

Defines information available through the Alto Operating System Swat resident.

ProcessDefs (Process) Processes and Monitors

Includes a number of extensions to the language facilities for processes, monitors, and condition variables. Priorities and timeouts can be adjusted, and processes can be detached and aborted.

RealDefs (not implemented) Miscellaneous

Defines the procedure calls generated by the compiler for operating on REAL data types. (Mesa does not provide an implementation of this interface.)

SegmentDefs (Segments, Files, Swapper) Segment and File Packages

Operations on data and file segments are contained here; this includes virtual memory management and swapping. Basic operations on files and their attributes are also included.

StreamDefs (KeyStreams, StreamsA, StreamsB) Disk, Display, Keyboard, and Streams Packages

Defines the operations common to all streams. It also includes some device-dependent operations unique to the disk, keyboard, and display.

StringDefs (StringsA, StringsB) String Package

A utility package for copying, comparing, and converting strings and substrings.

SystemDefs (FSP, Segments) Storage Management

Provides a simplified interface to the segment and free storage package for allocating and releasing temporary storage.

TimeDefs (TimeConvert) Time Package

Includes a number of procedures for converting between internal (32 bit GMT), intermediate (unpacked record) and external (string) time formats.

TrapDefs (Resident, NonResident) Traps

Defines the runtime implementation of traps generated by the hardware and software.

3. System Organization

Mesa systems are available in both *standard* and *basic* configurations. The former is intended for the day to day operation of most program developers, who do not want to provide a large amount of software in order to perform simple tasks. On the other hand, the basic system is intended for clients who are building complete applications from the ground up, and wish to replace many of the standard facilities with their own versions. Several optional packages are available for augmenting the capabilities of the the standard and basic systems in various directions, and we hope that many more will be added.

Standard System

Except for the optional packages listed below, the standard Mesa system (MESA.IMAGE) includes all the facilities described in this document. In addition to the interfaces described in the previous section, it includes a simple user interface called the *Mesa Executive*; the commands it accepts are described in **Section 4.**

Basic System

The basic system (BASICMESA.IMAGE) includes the following facilities: runtime support for the language: **Signaller, Resident, NonResident**; modules used to access files on the disk: **Segments, Files, Swapper, DiskIO, DiskKD, BFS, StreamsA, StreamsB, Directory**; the process package (**Process**), the string package (**StringsA, StringsB**), and the free storage package (**FSP**); the loader and its associated modules; a debugger interface; an interrupt key; and a nub of the *Mesa Executive.* The are no facilities for using the display or keyboard in the basic system, nor are there any of the modules associated with making image files. These and other packages are available as separate configurations (see below). See BASICMESA.CONFIG for the list of exported interfaces.

Optional Packages

To assist in tailoring applications built using the basic system, some facilities are optional and packaged separately. The configurations marked as standard are included in the standard Mesa system; the others may be loaded optionally. None of these packages are part of the basic system; they must be included in the user's configuration (or otherwise loaded) if they are needed.

CheckPoint

Implements check files (**Image Files**).

DisplayPackage (STANDARD)

Implements the display and font procedures. An instance of **StreamIO** is also included (**Display Package**).

ImageMaker (STANDARD)

Implements image files (**Image Files**).

ImageRunner

Implements image file loading (**Image Files**).

UnNewConfig

Implements unloading of configurations (**Modules**).

4. Mesa Executive

The *Mesa Executive* serves as the user interface for the standard Mesa system. It provides facilities for loading and starting Mesa programs, and for saving the state of a system in an image file; it is also the primary interface between your program and the Mesa debugger. The *Mesa Executive* has five commands, each identified by their first letter: `New`, `Start`, `Debug`, `MakeImage`, and `Quit`. The commands are discussed in detail below.

`New` **[filename]**

Performs a NEW on the configuration (or module) **filename**, loading it into the system. If no extension is supplied, "`.bcd`" is assumed. The global frame of the control module of the configuration is returned, but it is not started. If the configuration contains no control module, zero is returned.

`Start` **[frame]**

Performs a START on **frame** (specified in octal). Note that module parameters can not be supplied. Typing ESC in place of **frame** starts the frame returned by the last `New` command.

`Debug [confirm]`

Invokes the Mesa Debugger. Typing ↑SWAT at any point during execution of a Mesa program also invokes the Debugger. In addition, ↑SHIFT-SWAT attempts to invoke Swat.

`MakeImage` **[filename]**

Saves the current state of the system (including any programs that have been loaded) on the image file **filename**. If no extension is supplied, "`.image`" is assumed. The section on **Image Files** contains further details on the `MakeImage` command.

`Quit [confirm]`

Exits the Mesa environment, cleaning up its state, and returns to the Alto Executive. Typing SHIFT-SWAT at any point during execution also attempts to abort the computation and return to the *Alto Executive.*

Note: on Alto II keyboards, FR5 (lower right key) also functions as a SWAT key.

The *Mesa Executive* keeps a typescript of these commands (and all data sent to the standard output stream) on the file MESA.TYPESCRIPT; it can be used as a log of your Mesa session.

Most of the above commands can be specified on the command line used to invoke Mesa (and some are valid for BasicMesa, as well). See the *Mesa User's Handbook* for further details.

5. System Facilities

The material which follows is divided into several subsections, each of which describes a more or less logically disjoint subset of the system. The subsections are listed below, and follow this section in alphabetical order. The relevant DEFINITIONS modules, which clients will want to reference, are named in parentheses below, as well as in the subsections which follow.

Directory Package (**DirectoryDefs**)
Disk Streams Package (**StreamDefs**)
Display Package (**AltoDisplay**, **DisplayDefs**, **FontDefs**, **StreamDefs**)
File Package (**AltoDefs**, **AltoFileDefs**, **SegmentDefs**)
Image Files (**ImageDefs**)
Keyboard Package (**KeyDefs**, **StreamDefs**)
Miscellaneous (**InlineDefs**, **MiscDefs**)
Modules (**FrameDefs**)
Processes and Monitors (**ProcessDefs**)
Segment Package (**AllocDefs**, **SegmentDefs**)
Storage Management (**FSPDefs**, **SystemDefs**)
StreamIO Package (**IODefs**)
Streams (**StreamDefs**)
String Package (**AltoDefs**, **StringDefs**)
Time Package (**TimeDefs**)
Traps (**TrapDefs**)

Alto/Mesa Directory Package

April 1979

The Mesa directory package provides a number of procedures to manipulate the standard Alto directory (SysDir) or any file which follows the format of the Alto directory (see **DirectoryDefs**). Currently, these routines do *not* support either sub-directories or file version numbers. This section depends heavily on the section on Files, and that section should be read before this. (See *Alto Operating System Reference Manual* for file structure details.)

The operations described below come in pairs. The operation containing the word **Directory** operates on the standard Alto directory (SysDir). The other operation of the pair operates on the directory specified by the disk stream handle **dir**.

The simplest operation provided is to enumerate all of the file entries in the directory:

```
EnumerateDirectory: PROCEDURE [proc: PROCEDURE [POINTER TO FP, STRING]
    RETURNS [BOOLEAN]];

Enumerate: PROCEDURE [
    dir: DiskHandle, proc: PROCEDURE [POINTER TO FP, STRING] RETURNS [BOOLEAN]];
```

> Calls **proc** once for each filename in the directory, passing it pointers to the file's **FP** and name. Processing terminates when **proc** returns TRUE or the end of the directory is reached.
>
> Fine point: these parameters are local to the enumeration procedure and must be copied if they are to be retained after the enumeration completes.

The following procedures may be of use to programmers implementing features beyond those provided by **NewFile** and **DestroyFile**.

```
DirectoryLookup: PROCEDURE [fp: POINTER TO FP, name: STRING, create: BOOLEAN]
    RETURNS [old: BOOLEAN];

Lookup: PROCEDURE [
    dir: DiskHandle, fp: POINTER TO FP, name: STRING, create: BOOLEAN]
    RETURNS [old: BOOLEAN];
```

> Looks up **name** in the directory. If an entry already exists, its file pointer is copied into **fp** and TRUE is returned, otherwise FALSE is returned. In addition, if **create** is TRUE, the file will be created (with one empty data page), and the new file pointer will be copied into **fp**.

```
DirectoryLookupFP: PROCEDURE [fp: POINTER TO FP, name: STRING]
    RETURNS [old: BOOLEAN];
```

```
LookupFP: PROCEDURE [dir: DiskHandle, fp: POINTER TO FP, name: STRING]
  RETURNS [old: BOOLEAN];
```

Similar to **DirectoryLookup**, except that the directory is searched for a matching **FP**. It returns TRUE if the file pointer was found. In addition it will supply the filename if **name** is not NIL.

```
DirectoryPurge: PROCEDURE [fp: POINTER TO FP, name: STRING]
  RETURNS [found: BOOLEAN];
```

```
Purge: PROCEDURE [dir: DiskHandle, fp: POINTER TO FP, name: STRING]
  RETURNS [found: BOOLEAN];
```

Removes the entry corresponding to **name** from the directory (it does *not* disturb the file pages pointed to by the **FP**, however). If the file is found, its file pointer is copied into **fp** and TRUE is returned, otherwise FALSE is returned.

```
DirectoryPurgeFP: PROCEDURE [fp: POINTER TO FP] RETURNS [found: BOOLEAN];
```

```
PurgeFP: PROCEDURE [dir: DiskHandle, fp: POINTER TO FP] RETURNS [found: BOOLEAN];
```

Similar to **DirectoryPurge**, except that the directory is searched for a matching **FP**. It returns TRUE if the file pointer was found, deleting the entry in the process. Perhaps it should also copy the file's name into a supplied parameter?

Alto/Mesa Disk Streams Package

April 1979

A disk stream (see **StreamDefs**) is an array-like representation of a disk file. Parts of the file may reside in memory from time to time at the convenience of the stream. Like most arrays, a stream has a *length*; unlike array variables, the length of a stream may be changed by appending to it, and the maximum length is very large. Disk streams are created by the procedures:

```
NewByteStream, NewWordStream: PROCEDURE [
    name: STRING, access: AccessOptions]
    RETURNS [DiskHandle];
```

A **FileHandle** for the file **name** is created with the given access and **DefaultVersion**. It is locked and opened, and a byte or word stream is attached to it. If **access** is **Append** only, the stream is positioned at the end of the file, otherwise at the beginning. If **access** is **DefaultAccess**, **Read** is assumed. If a valid **FileHandle** already exists, a stream may be attached to it by calling the procedures:

```
CreateByteStream, CreateWordStream: PROCEDURE [
    file: FileHandle, access: AccessOptions]
    RETURNS [DiskHandle];
```

The stream's **FileHandle** and access may be read directly from the **StreamObject** (after discrimination) using the field names **file** and **read**, **write**, **append**.

Disk streams are chained together using the link field in a **StreamObject**. The head of the list is returned by the procedure:

```
GetDiskStreamList: PROCEDURE RETURNS [DiskHandle];
```

The operations allowed on the stream's length are determined by its access options; these options are negotiated with the underlying file system (see the section on **Files**). The options supported by the stream package are:

Read: the length is a constant.
Write: the length may decrease.
Append: the length may increase.

A disk stream has as part of its state a current **index** into the array representation of the file. The first data item is at index zero, the last at **length**-1. An invariant of a disk stream is **index** <= **length**. The current index and length are used in defining the semantics of the standard operations on a disk streams. See the section on **Streams** for more information on these generic operations. For a disk stream, **s** (where **i** is an item of the appropriate type) they are:

reset[s]

Effect: sets the **index** to zero.

get[s]

If: **read** AND **index** < **length**.
Effect: **t** ← **s[index]**; **index** ← **index** + 1; RETURN[**t**].

putback[s, i]

Effect: **StreamOperation** error.

put[s, i]

If: **(write** AND **index** < **length)** OR
(append AND **index** = **length)**.
Effect: **s[index]** ← **i**; **index** ← **index** + 1; **length** ← MAX[**index**, **length**].

endof[s]

Effect: RETURN[**index** = **length**].

destroy[s]

Effect: IF ~**read** AND **index** # 0 THEN **length** ← **index** (i.e. truncate the file if it is not positioned at the beginning). Release the **FileHandle** if there are no segments attached to it.

Since output to the disk is buffered, there may be items in the stream that have not yet been written out. The **destroy** operation causes these items to be written out before releasing the **StreamObject**.

Actually, there is a little more to it. Disk streams deliver either byte or word items: in either case, the **index** is always computed in bytes. So the description above is a simplification of what really happens. Rather than clutter it up, suffice it to say that when accessing files in word mode, index values are always rounded up to word boundaries.

If it is necessary to truncate a file in the cases not covered by **destroy** (i.e. **read** OR **index** = 0), call

TruncateDiskStream: PROCEDURE [**stream: StreamHandle**];

Effect: **length** ← **index**; **stream.destroy[stream]**.

If it is necessary to assure that all the data in the stream has been written to the disk without releasing the **StreamObject**, call

CleanupDiskStream: PROCEDURE [**stream: StreamHandle**];

In addition to the standard operations, the following disk dependent functions are provided to efficiently copy large blocks of words to or from the stream:

ReadBlock: PROCEDURE [**stream: StreamHandle, address:** POINTER, **words:** CARDINAL]
RETURNS [**count:** CARDINAL];

If: **read**.
Effect: **count** ← MIN[**words,length-index**];

```
        FOR index IN [index..index + count) DO
          address↑ ← stream[index];
          address ← address + 1;
          ENDLOOP.

WriteBlock: PROCEDURE [stream: StreamHandle, address: POINTER, words: CARDINAL]
    RETURNS [count: CARDINAL];

        If: (write AND index < length) OR
          (append AND index = length).
        Effect: count ← IF append
            THEN words
            ELSE MIN[words,length-index];
          FOR index IN [index..index + count) DO
            stream[index] ← address↑;
            address ← address + 1;
            ENDLOOP.
          length ← MAX[index, length].
```

When using **ReadBlock** and **WriteBlock**, the initial **index** must be on a word boundary (otherwise the **StreamPosition** error results). Note that the returned value may be less than **words** if the stream's **access** does not allow reading or writing of the whole block (the **StreamAccess** error is *never* raised by either of these procedures).

The stream index alluded to above is actually a structure:

```
StreamIndex: TYPE = RECORD [
    page: PageNumber,
    byte: CARDINAL];
```

The first data byte of a stream is at **StreamIndex[0, 0]**. The current stream position can be determined by calling

```
GetIndex: PROCEDURE [stream: StreamHandle] RETURNS [StreamIndex];
```

It is quite acceptable to do double precision arithmetic on a **StreamIndex** (and even single precision operations on the individual fields, if you are careful about borrows, carries, and overflows). Note, however that a **StreamIndex** is not compatible with the double precision arithmetic of LONG INTEGERs. The paged structure of the index can be restored by invoking

```
NormalizeIndex: PROCEDURE [index: StreamIndex] RETURNS [StreamIndex];
```

It returns an index whose **byte** field is in the range **[0..CharsPerPage)**. An index may be modified by calling

```
ModifyIndex: PROCEDURE [index: StreamIndex, change: INTEGER]
    RETURNS [StreamIndex];
```

The current index may be set by calling

```
SetIndex: PROCEDURE [stream: StreamHandle, index: StreamIndex];
```

Note that this may actually extend the file (with unspecified data) if **Append** access is allowed. To determine if this will happen, you might first want to call

```
FileLength: PROCEDURE [stream: StreamHandle] RETURNS [StreamIndex];
```

Note that **FileLength** sets the stream to the end-of-file and returns the length as seen through the stream; this may differ from the physical length of the disk file (if, for example, items have been appended to the stream but not yet written to the disk).

You may test for the greater-than relation between two stream indexes by calling the procedures

```
GrEqualIndex: PROCEDURE [i1, i2: StreamIndex] RETURNS [BOOLEAN];
```

```
GrIndex: PROCEDURE [i1, i2: StreamIndex] RETURNS [BOOLEAN];
```

If a physical disk location is required along with the stream position, a file address (**FA**) will prove useful (see **AltoFileDefs**); it is similar to a **StreamIndex** with a disk address (**DA**) tacked on the front, except that the **page** field is one origin (in the Alto file system, page zero is the leader page).

```
FA: TYPE = MACHINE DEPENDENT RECORD [
    da: DA,
    page: PageNumber,
    byte: CARDINAL];
```

You may record the current stream position and re-establish it later, in a fashion similar to **GetIndex** and **SetIndex**, by calling the procedures

```
GetFA: PROCEDURE [stream: StreamHandle, fa: POINTER TO FA];
```

```
JumpToFA: PROCEDURE [stream: StreamHandle, fa: POINTER TO FA];
```

The special thing about **JumpToFA** is that the disk address in the **fa** is taken as a hint; if it doesn't work out (the page number or file serial number doesn't match the stream's version of them), **JumpToFA** will attempt to find the requested page via the shortest route and correct the **fa** accordingly. This may involve starting over at the beginning of the file. If that fails,

```
InvalidFP: SIGNAL [fp: POINTER TO FP];
```

will result, probably indicating that the file has been moved (or worse, deleted!) since the stream was attached to it. A call on some directory searching procedure may prove useful in this situation, to determine if retrying the operation (with a new **fp**) is appropriate.

Alto/Mesa Display Package

April 1979

The Mesa Display Package provides a simple, Teletype style interface to the Alto display (see **AltoDisplay**). There is provision for using any available font: however the display is restricted to a single font for any particular incarnation. The font operations (described at the end of this document) are independent of the display implementation and may be used by any other display package. The package will optionally maintain a typescript of displayed output.

Display Stream

Normal access to the display is through a stream interface (see the section on **Streams**). There is no provision for multiple display streams. The module **SystemDisplay** implements the following procedures defined in **StreamDefs**:

```
GetDefaultDisplayStream: PROCEDURE RETURNS [DisplayHandle];
```

The interpretation of the basic stream operations is:

> **reset** clears the display and resets the typescript.
> **put** displays the character at the next sequential location.
> **get**, **putback** and **destroy** SIGNAL **StreamError[StreamAccess]**.
> **endof** returns FALSE.

In addition, the following operations are defined for display streams:

> **clearCurrentLine: PROCEDURE [stream: StreamHandle]** clears the current line of the display. The next character will be displayed at the left margin. The typescript is repositioned to the beginning of the line.
> **clearLine: PROCEDURE [stream: StreamHandle, line: CARDINAL]** clears **line** on the the display. The next character will be displayed at the left margin of that line. The typescript is repositioned to the beginning of the line. In the current Alto/Mesa implementation, **clearLine** is a no-op.
> **clearDisplayChar: PROCEDURE [stream: StreamHandle, char: CHARACTER]** erases the last character written on the display. The character must be supplied since the stream retains no knowledge of what characters are displayed (the typescript is optional).

DisplayDefs defines some additional interface procedures:

```
InitDisplay: PROCEDURE [
    dummySize, textLines, nPages: CARDINAL, f: FontDefs.FontHandle];
```

> This procedure initializes the display with **dummySize** blank scan lines at the top and room for at most **textLines** lines of text using **nPages** pages of memory for data structures and bitmap. The number of text lines and the display width are reduced if necessary to make everything fit in **nPages**.
>
> FinePoint: the amount of memory necessary to guarantee that n full width lines of text can be displayed is

n*(4+h*w) words, where h is the height of the font in scan lines (rounded up to an even number) and w is the width of the display in words.

SetSystemDisplaySize: PROCEDURE **[nTextLines, nPages:** CARDINAL**];**

Clears the display and reinitializes it with the new parameters.

SetSystemDisplayWidth: PROCEDURE **[indent, width:** CARDINAL**];**

Clears the display and reinitializes it with the new width parameters. **indent** is the number of bits from the left edge of the screen to the first display position and **width** is the width of a display line in bits. (The actual width will be the nearest multiple of 32 bits.).

FinePoint: indenting by multiples of 16 bits is very efficient.

SetDummyDisplaySize: PROCEDURE **[nScanLines:** CARDINAL**];**

Changes the size of the blank space at the top of the display. The space will be rounded up to an even number of scan lines and may be zero.

Background: TYPE **= {white, black};**

DisplayOff: PROCEDURE **[color: Background];**

DisplayOn: PROCEDURE**;**

DisplayOff releases all of the space allocated to the display and swaps out the font. All of the parameters of the display are saved so that **DisplayOn** can restore the previous state (but not the contents) of the display.

StartCursor: PROCEDURE**;**

StopCursor: PROCEDURE**;**

StartCursor forks a process in **DisplayControl** that blinks the cursor. If the cursor process is already running, it is a no op. **StopCursor** stops it.

BlinkCursor: PROCEDURE RETURNS **[**BOOLEAN**];**

Blinks a "cursor" at the position where the next character will be displayed. Each call changes the state of the cursor from "on" to "off" or vice versa. **BlinkCursor** returns TRUE if the last call changed the cursor state to on. The cursor is always turned off before a character is displayed or erased.

SetTypeScript: PROCEDURE **[StreamDefs.DiskHandle];**

This procedure establishes a disk stream as a typescript for the display. Passing NIL will disable the typescript. Characters sent to the display stream while the display is off will appear in the typescript. The typescript must be an open byte stream with **Read + Write + Append** access.

GetTypeScript: PROCEDURE RETURNS **[StreamDefs.DiskHandle];**

This procedure returns the disk stream that is behind the typescript for the display.

```
DisplayControl: PROGRAM;
```

This is the control module used in Mesa.image. It will initialize the display, font and typescript (using either MesaFont.al or SysFont.al and Mesa.Typescript) and start a process to call **BlinkCursor** at half second intervals. It will also reestablish the display (including font and typescript) after a MakeImage or MakeCheckPoint.

Fonts

A **FontObject** provides a simple object style interface to character fonts. Operations are provided for painting or erasing characters from the font in a bitmap. **FontDefs** defines the following TYPEs and PROCEDURES:

```
BitmapState: TYPE = RECORD [
    origin: POINTER,
    wordsPerLine, x, y: [0..77777B)];
```

A **BitmapState** describes where a character will be placed within a bitmap. **origin** is a POINTER to the beginning of the bitmap. **wordsPerLine** is the horizontal width of the bitmap (it must be even if the bitmap is to be displayed). **x** and **y** are measured from the upper left corner in bits right and scan lines down respectively.

```
FontHandle: TYPE = POINTER TO FontObject;

FontObject: TYPE = RECORD [
    paintChar: PROCEDURE [FontHandle, CHARACTER, POINTER TO BitmapState],
    clearChar: PROCEDURE [FontHandle, CHARACTER, POINTER TO BitmapState],
    charWidth: PROCEDURE [FontHandle, CHARACTER] RETURNS [CARDINAL],
    charHeight: PROCEDURE [FontHandle, CHARACTER] RETURNS [CARDINAL],
    close: PROCEDURE [FontHandle],
    destroy: PROCEDURE [FontHandle],
    lock: PROCEDURE [FontHandle] RETURNS [POINTER],
    unlock: PROCEDURE [FontHandle]];
```

A **FontObject** implements the following operations:

paintChar: ORs the specified character from the font into the bitmap position specified in the BitmapState; x is updated to point to the next character position. There is no bounds checking.

clearChar: erases the bit rectangle which bounds the character. The input state points just beyond the character and is modified to point to where the character used to be. **paintChar[f, c, s]** followed by **clearChar[f, c, s]** leaves **s** unchanged.

charWidth, **charHeight**: return the width and height of a character in bits and scan lines respectively.

close: swaps the font out of memory if it is not otherwise in use. The font will always be swapped in when needed. It is not generally locked.

destroy: calls **close** and then releases the space allocated for the **FontObject**. The font segment is not deleted.

lock: locks the font segment in memory and returns a POINTER to the first word. This can be used to implement other operations on the bits in the font. Note that nothing in a **FontObject** dictates what font format is used.

unlock: unlocks the font after a call to **lock**.

CharWidth: PROCEDURE **[font: FontHandle, char:** CHARACTER**]** RETURNS **[**CARDINAL**];**

Equivalent to **font.charWidth[font, char]**.

CharHeight: PROCEDURE **[font: FontHandle, char:** CHARACTER**]** RETURNS **[**CARDINAL**];**

Equivalent to **font.charHeight[font, char]**.

CreateFont: PROCEDURE **[SegmentDefs.FileSegmentHandle]** RETURNS **[FontHandle];**

Allocates space (from the system heap) for a **FontObject** and initializes its operations to use the font in the supplied segment.

GetFont: PROCEDURE RETURNS **[FontHandle];**

Returns the **FontHandle** for the system font.

The module **AlFont** implements **FontObject**s for "Al" format fonts. Modules for other font formats can be substituted easily. At this time no other modules have been written or planned.

Mesa.image uses **SystemDisplay**, **AlFont**, and **DisplayControl** for its default display. They are also available as a separate package in **DisplayPackage** for users of BasicMesa. **DisplayPackage** also contains an instance of **StreamIO**.

Alto/Mesa File Package

April 1979

Logically, the Mesa file package is a sub-module of the segmentation machinery, but it is described separately because other objects (e.g. disk streams) also use this interface. Internally, the file machinery maintains a set of items called **FileObjects**: a file object, among other things, contains the file's disk address and serial number, as well as its access rights, several reference counts, and an optional file length hint.

The Mesa system follows most conventions of the Alto file system (although some, including multiple versions and sub-directories are not currently supported.) See the *Alto Operating System Reference Manual* document for a description of the Alto file system. A description of the various procedures used to manipulate the Alto's directory appears in the section on **Directories**.

Files

A file is an integral number of pages which logically appear to be contiguous, irrespective of their physical location. The pages of a file are numbered from zero up to some maximum (see **AltoDefs**):

```
MaxFilePage: CARDINAL;   -- maximum file page number

PageNumber: TYPE = [0..MaxFilePage];
```

In the Alto file system, page zero of the file (the leader page) is special; it contains file status information. Thus the data actually begins at page one.

Externally, a file is known by its name, which is just a string. Internally, Mesa retains only a file's **FP**, which is an abbreviated form of the Alto file system's file pointer (see **AltoFileDefs**):

```
FP: TYPE = RECORD [
    serial: SN,          -- internal file serial number
    leaderDA: vDA];      -- first virtual disk address
```

The correspondence between file names and **FP**s is maintained in the file system's directory (SysDir). After the file is initially looked up, the name is discarded; the Mesa world deals only in **FP**s thereafter. A directory search is required if the name must be recovered.

File Objects

A **FileHandle** is used to refer to a file in the Mesa environment, and can be obtained by a call on **NewFile** (described bedow); it is simply a pointer to a record called a **FileObject** (see **SegmentDefs**).

```
FileHandle: TYPE = POINTER TO FileObject;

FileObject: TYPE = RECORD [
   open: BOOLEAN,                    -- if the file is open
   read, write, append: BOOLEAN,     -- access rights
   lock: [0..MaxLocks],              -- reference count
   segcount: [0..MaxSegs],           -- attached segments
   swapcount: [0..MaxRefs],          -- swapped in segments
   . . . ];                          -- plus other private fields
```

The following options are used when creating new file objects:

```
AccessOptions: TYPE = [0..7];
   Read: AccessOptions = 1;
   Write: AccessOptions = 2;
   Append: AccessOptions = 4;

VersionOptions: TYPE = [0..3];
   NewFileOnly: VersionOptions = 1;
   OldFileOnly: VersionOptions = 2;
```

Read access allows existing pages of the file to be read; **Write** means that existing pages can be written (or deleted; perhaps a separate **Delete** option should be included). **Append** allows new pages to be added to the end of the file (files do not have holes in them).

Fine point: **Append** does *not* imply **Write** access. **Append** means that new pages may be added to the file but existing pages may not be modified.

Disallowed combinations are {**NewFileOnly**, **OldFileOnly**} and {**NewFileOnly**, ~**Append**}. If **Append** access is not specified, **OldFileOnly** is assumed. If you like, you may specify **DefaultAccess**, which is equivalent to **Read**. (Note that **Append** access must be specified in order to create the file.)

Fine point: if **DefaultVersion** is specified, the file is created if it did not previously exist.

Signals

Signals associated with **FileObject**s are as follows:

```
FileNameError: SIGNAL [name: STRING];
```

The file name is invalid, or the file does not exist (**OldFileOnly**), or the file does exist (**NewFileOnly**).

```
FileAccessError: SIGNAL [file: FileHandle];
```

An attempt to perform some operation not allowed by the current access, or the requested access and version options are inconsistent (see the disallowed combinations above).

```
InvalidFP: SIGNAL [fp: POINTER TO FP];
```

A file positioning operation has determined that the file serial number in the **FP** of the file object does not match the disk label. Most likely, the **FileHandle** references a file which has been moved or destroyed (or clobbered) since it was last referenced

```
FileError: SIGNAL [file: FileHandle];    -- all other file errors
```

File Creation/Deletion

A **FileObject** is created using the following procedures:

```
NewFile: PROCEDURE [
    name: STRING, access: AccessOptions, version: VersionOptions]
    RETURNS [FileHandle];
```

Given a file name and access rights, this procedure creates a new file object and returns a pointer to it. A check is made that the file exists in the directory, creating it if necessary, but the file is not opened as a result of this call. Objects attached to the file (segments and streams, for example) ensure that the file is open before attempting a transfer. If there is already a file object for the file specified, its access is updated (by or'ing; this is not a protection system), and a pointer to the existing object is returned.

```
InsertFile: PROCEDURE [fp: POINTER TO FP, access: AccessOptions]
    RETURNS [FileHandle];
```

Creates a file object directly from **fp**, without searching the directory. If there is already a file object with a matching **fp**, its access is updated (by or'ing; this is not a protection system), and a pointer to the existing object is returned.

Internally, Mesa keeps track of the number of segments attached to each file (**segcount**) and of those the number which are currently swapped in (**swapcount**). When the **swapcount** goes to zero, the file may be closed, and when the **segcount** goes to zero, the file object is released (only the latter operation happens automatically). Since a file may have other objects attached to it (streams, for example), it may be necessary to prevent the file object from being released even when there are no more segments attached to it. The **lock** field serves this purpose, and is manipulated by the procedures

```
LockFile, UnlockFile: PROCEDURE [file: FileHandle];
```

A maximum of **MaxLocks** locks may be performed on each file object. Note that a file object is *not* automatically released when its lock count goes to zero.

A **FileObject** is released by

```
ReleaseFile: PROCEDURE [file: FileHandle];
```

The file is first closed if it is open; then its file object is released. A **FileError** will be generated if there are segments associated with the file at the time of this call. *Note:* except for this error check, releasing a file which is locked is a no-op.

A file is physically destroyed by calling

DestroyFile: PROCEDURE **[file: FileHandle];**

In addition to releasing the file object, the file's pages are deleted and its entry is removed from the directory. The file object must not have any segments currently attached to it, nor may it be locked; either condition results in a **FileError**.

To be on the safe side, destroying a file is somewhat complicated if it currently has segments attached to it. The file must first be locked, then all of its segments deleted and all streams attached to it destroyed, then the file should be unlocked and finally **DestroyFile** should be called. This sequence assumes that no other client has a lock on the file.

File Properties

Characteristics of the disk file associated with a **FileObject** are obtained and changed using the following procedures:

FindFile: PROCEDURE **[fp:** POINTER TO **FP]** RETURNS **[FileHandle];**

Searches all existing file objects for one whose serial number and disk address match those contained in **fp**. Returns NIL if no match can be found.

GetFileFP: PROCEDURE **[file: FileHandle, fp:** POINTER TO **FP];**

Copies the file pointer from **file** into **fp**.

GetFileAccess: PROCEDURE **[file: FileHandle]** RETURNS **[access: AccessOptions];**

Converts the **read**, **write**, and **append** bits of a file object into a form that can be passed to **NewFile**.

SetFileAccess: PROCEDURE **[file: FileHandle, access: AccessOptions];**

Or's **access** into the file object (this is not a protection system).

File lengths hints are no longer contained in every **FileObject**. A separate object contains the length for a file. This separate length object is not required and is allocated only when operations on file lengths are invoked (see **SegmentDefs**). Operations on file lengths are:

GetEndOfFile: PROCEDURE **[file: FileHandle]**
RETURNS **[page: PageNumber, byte:** CARDINAL**];**

Returns the page number of the last page in the file that contains data, together with the number of bytes in that page (the number of the first non-existent byte in the page, counting from zero). In the Alto file system, page zero is the leader page, the first data page being page one. Note that if the last data page is full, a null page is appended to the file, but **GetEndOfFile** does *not* tell you about it (so do not count on it being there). For an empty file, this routine returns **[0, BytesPerPage]** (reflecting the existence of the leader page).

GetEndOfFile first opens the file (if it is closed) to obtain the length hint from the leader page. It also inserts the current file length into the file length object (creating one if necessary), so that subsequent requests for the length will not require reading the disk.

SetEndOfFile: PROCEDURE **[file: FileHandle, page: PageNumber,** byte: CARDINAL];

Extends or truncates the file as necessary to make **page** the number of its last data page, with **byte** bytes in it. The arguments are first adjusted to include a null page if **byte = BytesPerPage.** Extending requires **Append** access, truncating requires **Write** access.

Miscellaneous

The procedure **EnumerateFiles** is provided so that one may conveniently scan all file objects that currently exist.

EnumerateFiles: PROCEDURE **[proc:** PROCEDURE **[FileHandle]** RETURNS **[**BOOLEAN**]]** RETURNS **[file: FileHandle];**

This procedure calls **proc** once for each file object that is currently exists. This process will terminate when the list of file objects is exhausted or when **proc** returns TRUE. In the latter case, the **FileHandle** of the last **FileObject** processed is returned. Otherwise, NIL is returned.

If new file objects are created while **EnumerateFiles** is in control, it is not guaranteed that they will be included in the sequence of **FileHandles** passed to **proc.**

Alto/Mesa Image Files

April 1979

A Mesa image file contains the information necessary to start execution of a Mesa system. In addition to image files that contain all code and data, there are also image files, called check files, that contain only data. (Check files know the addresses of other files on the disk and therefore cannot be moved like other image files). This section defines the facilities provided to make them. See **ImageDefs** for further details.

Making Image Files

The standard system contains code to make an image file of itself and any user programs which have been loaded. Clients may make an image file by calling the procedure:

MakeImage: PROCEDURE **[name:** STRING**];**

> Make an image file on file **name**. Returns to Alto Executive. When the image file is restarted, **MakeImage** returns to its caller.

Clients may also make an image file by invoking the Mesa Executive's MakeImage command. It accepts a filename, defaults the extension to ".IMAGE" and calls **MakeImage [name]**. When the image file is restarted, the Mesa Executive will be ready to accept a new command.

MakeImage normally merges all the BCDs that have been loaded into one bcd, and cleans up system data structures. This results in faster loading of additional BCDs into the new image.

Those clients that do not want BCDs merged during a **MakeImage** because they call **UnNewConfig** may use the following procedure:

MakeUnMergedImage: PROCEDURE **[name:** STRING**];**

Signals that may be generated by **MakeImage** or **MakeUnMergedImage** are:

InvalidImage: SIGNAL**;**

> An attempt has been made to make an image file on top of the currently excuteing image file.

NoRoomInImageMap: SIGNAL**;**

> The **map** in the **ImageHeader** has filled up. This usually means that there are too many segments in memory at the time the image file is made.

Check Files

Check files differ from normal image files in that they do not contain all the code and data to start the execution of the image file. Check files only contain the data of the Mesa system, and all code and other segments remain in their original files. As a result, check files are made and restarted very quickly. However, care must be taken not to destroy files that are pointed to by the check file. Check files are useful for making check points in the execution of a Mesa system. The procedures and signals that make check files are:

MakeCheckPoint: PROCEDURE **[name:** STRING] RETURNS **[restart:** BOOLEAN];

> Makes a check file on file **name**. Returns to caller when finished as if nothing happened with **restart** FALSE. If the check file is being restarted, it returns to the caller with TRUE.

NoRoomInCheckMap: SIGNAL;

> In **MakeCheckPoint**, the **map** in the **ImageHeader** has filled up. This usually means that there are too many segments in memory at the time the check file is made.

The ability to make check files is not included in the basic system. Clients should include the **CheckPoint** module in their configuration.

Running Image Files

Mesa programs may run another image file without returning to the Alto Executive. The procedures and signals that implement this are:

RunImage: PROCEDURE **[file: SegmentDefs.FileSegmentHandle]**;

> Runs the image file specified by **file** where **file** is a segment handle for the header of the image file. The current state of the present image file is lost, and the new image file is loaded and started.
>
> Fine point: currently the header of an image file is page one (this may change in the future, and may not be constant). Communication between the two image files must be done via disk files (like Com.cm). See the *Alto Operating System Reference Manual.*

InvalidImage: SIGNAL;

> In **RunImage**, **file** specifies an invalid image file.

NoRoomForLoader: SIGNAL;

> In **RunImage**, there is no room for the bootstrap loader that loads and starts the image file.

Clients should include the configuration **ImageRunner** in their configuration.

Miscellaneous

The version stamp of the currently running image file may be obtained by calling the procedure:

```
ImageVersion: PROCEDURE RETURNS [BcdDefs.VersionStamp];
```

The time that the currently running image file was created may be obtained by calling the procedure:

```
ImageTime: PROCEDURE RETURNS [TimeDefs.PackedTime];
```

The same information can be obtained about the caller's BCD from the procedures:

```
BcdVersion: PROCEDURE RETURNS [BcdDefs.VersionStamp];

BcdTime: PROCEDURE RETURNS [TimeDefs.PackedTime];
```

A client may stop execution of a Mesa system and return to the Alto Executive by calling:

```
StopMesa: PROCEDURE;
```

A client may terminate execution of a Mesa system as if shift-swat had been typed and return to the Alto Executive by calling:

```
AbortMesa: PROCEDURE;
```

Cleanup Procedures

Client programs sometimes need to be notified when certain events occur so they can perform various cleanup functions. Cleanup procedures provide a facility to notify client programs when image files are made or restarted, and when stopping execution of Mesa programs. The types and data structures involved with Cleanup procedures are:

```
CleanupProcedure: TYPE = PROCEDURE [why: CleanupReason];

CleanupItem: TYPE = RECORD [
   link: POINTER TO CleanupItem,
   mask: CARDINAL,
   proc: CleanupProcedure];

CleanupReason: TYPE = NovaOps.CleanupReason;

NovaOps.CleanupReason: TYPE = {
   Finish, . . . Save, Restore, CheckPoint, Restart, Continue, . . .};
```

CleanupReasons are copied from **NovaOps**. Valid ones are:

Finish -- **StopMesa** was called.
Save -- **MakeImage** was called.
Restore -- returning from **MakeImage**.
Checkpoint -- **Checkpoint** was called.
Restart -- returning from **Checkpoint** restarted by the Alto Executive.
Continue -- returning from **Checkpoint**.

The following procedures may be used to add or remove a cleanup procedure. The list of **CleanupItem**s is processed in the opposite order when returning from an image file than when making one.

```
AddCleanupProcedure: PROCEDURE [POINTER TO CleanupItem];

RemoveCleanupProcedure: PROCEDURE [POINTER TO CleanupItem];
```

Warning: these procedures may not be invoked from inside Cleanup procedures.

```
CleanupMask: ARRAY CleanupReason OF CARDINAL =
   [1, 2, 4, 10B, 20B, 40B, 100B, 200B, 400B, 1000B];
```

When the Cleanup procedure mechanism is invoked, each item is examined. If the bit corresponding to the reason is set in **item.mask**, **item.proc** is called. Clients must set the mask field so that their procedure is invoked only for the reasons they expect. For example, if a client needed to be notified when a normal image file was being made or started, the mask would be set to:

```
CleanupMask[Save] + CleanupMask[Restore].
```

Warning: if the **item.mask** is improperly set, **item.proc** may be invoked at sensitive times, probably resulting in erroneous behavior.

MakeImage Restrictions

1. The name of the new image file may not be the same as the name of the image file running at the time **MakeImage** is called. An image file can be renamed any time it is not running.

2. The user program should not have handles on any files or disk streams. Any file segments allocated by a user program will be made a part of the new image file.

3. This list of restrictions may not be exhaustive. In general you should avoid doing anything other than loading modules and initializing data structures before making an image file.

MakeCheckPoint Restrictions

1. The name of the new image file may be the same as the name of the image file running at the time **MakeCheckPoint** is called, only if the current image file is a check file. A check file should not be renamed.

2. The files that are pointed to by the check files should not be moved or altered. Care must be taken with open disk streams that the pages of the current position of the stream are not moved or destroyed.

3. This list of restrictions may not be exhaustive and care must be taken with all files that are open at the time the check file was make.

CleanupProcedure Restrictions

1. **CleanupProcedure**s are called with interrupts and timeouts turned off.

2. This list of restrictions may not be exhaustive and care must be taken when using **CleanupProcedure**s.

Alto/Mesa Keyboard Package

April 1979

The Keyboard package consists of the modules **KeyStreams** and **Keyboard** and provides a Teletype style interface to the undecoded keyboard through a device independent stream interface. Multiple, independent keyboard streams are supported. A default keyboard stream is created at initialization time. (See the section on **StreamIO** for higher level operations.) The Keyboard Package also optionally causes the hardware cursor to track the mouse. A single keyboard process runs at interrupt level approximately 60 times per second to sample the keyboard hardware.

The following procedures and types are defined in **StreamDefs**.

KeyboardHandle: TYPE = POINTER TO **Keyboard StreamObject;**

The standard operations on a keyboard stream are:

> **reset[s]** clears the buffer associated with **s**; any characters in the buffer are lost.
>
> **get[s]** returns the next character in the buffer; if **endof[s]** is TRUE, WAITs until it is FALSE.
>
> **putback[s,i]** modifies the stream so that the next **get[s]** will return **i**, independent of any type-ahead. If the buffer is full, **putback** is a no-op (sorry about that).
>
> **put[s,i]** produces a **StreamAccess** error.
>
> **endof[s]** TRUE if there are no characters in the buffer.
>
> **destroy[s]** destroys **s** in an orderly way, freeing the space it occupies. Any characters in the buffer at the time of the **destroy** are lost. If **s** is the current keystream, the **StreamOperation** error results.

CreateKeyStream: PROCEDURE RETURNS **[KeyboardHandle];**

> Creates a new keyboard streams. Any number of streams may be created. Each stream has its own buffer for type ahead. (Space for the **StreamObject** is allocated from the system heap.)

GetDefaultKey: PROCEDURE RETURNS **[KeyboardHandle];**

> Returns the default stream (created at initialization time).

GetCurrentKey: PROCEDURE RETURNS **[KeyboardHandle];**

> Returns the current stream. Initially the default keyboard stream is current.

OpenKeyStream: PROCEDURE **[stream: StreamHandle];**

Makes **stream** the current keyboard stream.The stream which was current before the call is undisturbed, except that input characters are no longer directed to it by the keyboard process, but to **stream** instead.

CloseKeyStream: PROCEDURE **[stream: StreamHandle];**

Makes the default keyboard stream current. Characters already typed remain in **stream**'s buffer. The **StreamOperation** error results if **stream** is not the current stream.

CursorTrack: PROCEDURE **[BOOLEAN];**

Calling this procedure with TRUE enables cursor tracking; FALSE disables it. When tracking is enabled, the mouse coordinates are copied to the cursor coordinates each time the keyboard process runs.

Low Level Access

The basic system does not provide access to the keyset or mouse through the stream. Definitions are provided for clients wishing to access the bits of the keyboard directly or to change the interpretation of any of the keys. The module **KeyDefs** defines types and procedures for lower level access to the keyboard hardware.

```
updown: TYPE = {down, up};

KeyBits: TYPE = MACHINE DEPENDENT RECORD [
    blank: [0..377B],      -- not used
    Keyset1, Keyset2, Keyset3, Keyset4, Keyset5: updown,
    Red, Blue, Yellow: updown,
    Five, Four, Six, E, Seven, D, U, V,
    Zero, K, Dash, P, Slash, BackSlash, LF, BS: updown,
    Three, Two, W, Q, S, A, Nine, I,
    X, O, L, Comma, Quote, RightBracket, Spare2, Spare1: updown,
    One, ESC, TAB, F, Ctrl, C, J, B,
    Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3: updown,
    R, T, G, Y, H, Eight, N, M,
    Lock, Space, LeftBracket, Equal, RightShift, Spare3, FL4, FR5: updown];

Keys: POINTER TO KeyBits = -- magic memory location -- ;

MouseButton: TYPE = {RedYellowBlue, RedBlue, RedYellow, Red, BlueYellow, Blue,
    Yellow, None};

MouseBits: TYPE = MACHINE DEPENDENT RECORD [
    blank: [0..377B],          -- Diablo, Versatec, etc.
    keyset: [0..37B],          -- 0 => down, i.e. normal state is 37B
    buttons: MouseButton];
```

```
Mouse: POINTER TO MouseBits = -- magic memory location -- ;

KeyName: TYPE = {
    . . . ,          -- unused values
    Keyset1, Keyset2, Keyset3, Keyset4, Keyset5,
    Red, Blue, Yellow,
    Five, Four, Six, E, Seven, D, U, V,
    Zero, K, Dash, P, Slash, BackSlash, LF, BS,
    Three, Two, W, Q, S, A, Nine, I,
    X, O, L, Comma, Quote, RightBracket, Spare2, Spare1,
    One, ESC, TAB, F, Ctrl, C, J, B,
    Z, LeftShift, Period, SemiColon, Return, Arrow, DEL, FL3,
    R, T, G, Y, H, Eight, N, M,
    Lock, Space, LeftBracket, Equal, RightShift, Spare3, FL4, FR5};
```

Alto II names for some keys are different.

```
    FL1: KeyName = DEL;
    FL2: KeyName = LF;
    BW: KeyName = Spare1;
    FR1: KeyName = Spare3;
    FR2: KeyName = BackSlash;
    FR3: KeyName = Arrow;
    FR4: KeyName = Spare2;

KeyItem: TYPE = RECORD [
    Letter: BOOLEAN,
    ShiftCode: [0..177B],
    NormalCode: [0..377B]];
```

There is a **KeyItem** for every key (including mouse and keyset keys). A **NormalCode** = 0 causes the key to be ignored; a **ShiftCode** = 0 puts in a zero for the key when the shift key is down; **Letter** means that the **ShiftCode** is selected by the shift lock key. Note that the **ShiftCode** is 7 bits and the **NormalCode** is 8 bits.

```
ChangeKey: PROCEDURE [key: KeyName, action: KeyItem]
    RETURNS [oldAction: KeyItem];
```

This procedure changes the meaning of a key and returns the old value.

Initialization

BasicMesa does not contain a keyboard package. Clients of BasicMesa who do not wish to supply their own keyboard procedures should include **Keyboard** and **KeyStreams** in their configurations. The following additional definitions from **KeyDefs** are needed:

```
Keyboard: PROGRAM;

KeyStreams: PROGRAM;
```

In order to initialize the keyboard process, a client must include these statements:

```
FrameDefs.MakeCodeResident[FrameDefs.GlobalFrame[KeyDefs.Keyboard]];
START KeyDefs.KeyStreams;
```

Alto/Mesa Miscellaneous

April 1979

This section describes some miscellaneous facilities in Alto/Mesa.

MiscDefs

```
Zero: PROCEDURE [p: POINTER, l: CARDINAL];

    FOR i IN [0..l) DO (p + i)↑ ← 0.

SetBlock: PROCEDURE [p: POINTER, v: UNSPECIFIED, l: CARDINAL];

    FOR i IN [0..l) DO (p + i)↑ ← v.

DAYTIME: PROCEDURE RETURNS [AltoFileDefs.TIME];
```

Returns the current time in the format maintained by the Alto Operating System and recorded in Alto files. This is *not* the same format as a **TimeDefs.PackedTime**.

```
CurrentTime: PROCEDURE RETURNS [LONG CARDINAL];
```

Returns the current time. This *is* the same format as a **TimeDefs.PackedTime**.

```
GetNetworkNumber: PROCEDURE RETURNS [CARDINAL];
```

Returns the number of the network to which the Alto is connected or 0 if there is no network or no response from a gateway.

Fine point: this procedure is used by the Compiler and others to generate unique identifiers for output files.

```
CommandLineCFA: PROCEDURE RETURNS [POINTER TO AltoFileDefs.CFA];
```

Returns a pointer to the **CFA** for the point at which the Mesa executive stopped reading the command line (COM.CM). This is valid only when a configuration is started from the command line using Mesa.image or BasicMesa.image. The caller can use the **CFA** to quickly open a stream on the command line and read any additional commands. The **CFA** should be updated before returning to the Mesa executive. See the sections on **Files** and **Streams** for more information.

```
CallDebugger: PROCEDURE [STRING];

WorryCallDebugger: PROCEDURE [STRING];
```

These procedures invoke the Debugger without the overhead of raising an uncaught signal. The Debugger will display the parameter if it is not NIL. The worry call will invoke the Debugger in worry mode.

InlineDefs

```
COPY: PROCEDURE [from: POINTER, nwords: CARDINAL, to: POINTER];

    FOR i IN [0..nwords) DO (to+i)↑ ← (from+i)↑.

DIVMOD: PROCEDURE [num, den: CARDINAL] RETURNS [quotient, remainder: CARDINAL];
```

Returns both the **quotient** and **remainder** of the *unsigned* division of **num** by **den**.

```
LDIVMOD: PROCEDURE [numlow: WORD, numhigh: CARDINAL, den: CARDINAL]
  RETURNS [quotient, remainder: CARDINAL];
```

Like **DIVMOD** except that the numerator is the double length *unsigned* number **numhigh*2^16 + numlow**. Results are undefined if the quotient is greater than 2^{16}-1.

```
LongNumber: TYPE = MACHINE DEPENDENT RECORD [
  SELECT OVERLAID * FROM
    lc => [lc: LONG CARDINAL],
    li => [li: LONG INTEGER],
    lu => [lu: LONG UNSPECIFIED],
    num => [lowbits, highbits: CARDINAL],
    ENDCASE];
```

The **LongNumber** structure permits access to the high and low words of LONG CARDINALS and LONG INTEGERS. Alternately, the following procedures may be used:

```
LowHalf: PROCEDURE [LONG UNSPECIFIED] RETURNS [UNSPECIFIED];

HighHalf: PROCEDURE [LONG UNSPECIFIED] RETURNS [UNSPECIFIED];
```

LowHalf and **HighHalf** allow access to the machine dependent components of LONG CARDINALS, LONG INTEGERS and LONG POINTERS. **LowHalf** returns the least significant word, while **HighHalf** returns the most significant word.

```
LowByte: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED];

HighByte: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED];
```

LowByte and **HighByte** return the least significant and most significant byte of a word respectively.

```
LongMult: PROCEDURE [CARDINAL, CARDINAL] RETURNS [product: LONG CARDINAL];
```

Returns the double precision result of the *unsigned* multiplication of the two single precision arguments.

```
LongDiv: PROCEDURE [num: LONG CARDINAL, den: CARDINAL] RETURNS [CARDINAL];
```

Returns the result of the *unsigned* division of **num** by **den**. The result is undefined if the quotient is greater than 2^{16}-1.

LongDivMod: PROCEDURE [**num:** LONG CARDINAL, **den:** CARDINAL]
RETURNS [**quotient, remainder:** CARDINAL];

Like **LongDiv** except both **quotient** and **remainder** are returned.

BITAND, BITOR, BITXOR: PROCEDURE [UNSPECIFIED, UNSPECIFIED] RETURNS [UNSPECIFIED];

These functions compute the bitwise AND, OR, or XOR of their arguments.

BITNOT: PROCEDURE [UNSPECIFIED] RETURNS [UNSPECIFIED];

Returns the ones complement of the input.

BITSHIFT: PROCEDURE [**value:** UNSPECIFIED, **count:** INTEGER] RETURNS [UNSPECIFIED];

Returns **value** shifted by ABS[**count**] bits. The shift is left if **count** > 0, right if **count** < 0.

Floating Point

While the Mesa system does not provide any support for floating point operations, the Compiler does recognize type REAL and generates calls to client supplied procedures via the system data vector (**SD**). The definitions of **SD** and all indices into **SD** are in **SDDefs**. In the following description the notation

SD[index]: PROCEDURE . . .

means that the client should declare a procedure **P** of the correct type and assign its descriptor to the appropriate element of **SD** (**SD[index]** ← **P**). The Compiler makes no assumptions about the representation of REALs except that they occupy two words.

SD[sFADD]: PROCEDURE [**a, b:** REAL] RETURNS [REAL];

Called to perform addition.

SD[sFSUB]: PROCEDURE [**a, b:** REAL] RETURNS [REAL];

Called to perform subtraction.

SD[sFMUL]: PROCEDURE [**a, b:** REAL] RETURNS [REAL];

Called to perform multiplication.

SD[sFDIV]: PROCEDURE [**a, b:** REAL] RETURNS [REAL];

Called to perform division (**a/b**).

SD[sFCOMP]: PROCEDURE [**a, b:** REAL] RETURNS [INTEGER];

Called to compare REALs. Return -1 if **a<b**, 0 if **a = b**, 1 if **a>b**.

SD[sFLOAT]: PROCEDURE [LONG INTEGER] RETURNS [REAL];

Called to convert fixed point to floating point.

SD[sFIX]: PROCEDURE [REAL] RETURNS [LONG INTEGER];

The Compiler does not generate calls to this procedure. It is included for completeness.

DoubleDefs

The following procedures from **DoubleDefs** have been retained for compatibility. They are no longer supported and clients should now use LONG CARDINALS. (**DAdd**, **DSub**, and **DCompare** are INLINE procedures and are available to any Mesa program. The others are available only to clients which include **Double** in their configuration.)

DAdd: PROCEDURE [a,b: LONG CARDINAL] RETURNS [LONG CARDINAL];

Returns the double precision *unsigned* sum of **a** and **b**..

DSub: PROCEDURE [a,b: LONG CARDINAL] RETURNS [LONG CARDINAL];

Returns the double precision *unsigned* difference of **a** and **b**.

Comparison: TYPE = {less, equal, greater};

DCompare: PROCEDURE [a,b: LONG CARDINAL] RETURNS [Comparison];

Returns **less** if **a<b**, **equal** if **a = b**. **greater** if **a>b**.

DMultiply: PROCEDURE [a,b: LONG CARDINAL] RETURNS [product: LONG CARDINAL];

Returns the product of the *unsigned* multiplication of **a** by **b**.

DDivide: PROCEDURE [num, den: LONG CARDINAL]
RETURNS [quotient, remainder: LONG CARDINAL];

Returns the quotient and remainder of the *unsigned* division of **num** by **den**.

DNeg: PROCEDURE [a: LONG CARDINAL] RETURNS [LONG CARDINAL];

While somewhat a contradiction in terms, this procedure negates a LONG CARDINAL by performing **DSub[0, a]**.

DInc: PROCEDURE [a: LONG CARDINAL] RETURNS [LONG CARDINAL];

DAdd[1, a].

AppendDouble: PROCEDURE [s: STRING, a: LONG CARDINAL];

The value of **a** is converted to text in decimal and appended to **s**.

Alto/Mesa Modules

April 1979

This section documents the operations required to manipulate modules at a more detailed level than provided by the language. It is intended for experienced programmers who have a genuine need to manipulate low level system structures. See **FrameDefs** for further details.

Global Frames

A **GlobalFrame** is the implementation of the PROGRAM language construct. All manipulation of **GlobalFrames** is done using **GlobalFrameHandles**.

```
GlobalFrameHandle: TYPE = POINTER TO GlobalFrame;

GlobalFrame: TYPE = RECORD [. . .];
```

The procedures which manipulate global frames are:

```
GlobalFrame: PROCEDURE [link: UNSPECIFIED] RETURNS [GlobalFrameHandle];
```

Returns the **GlobalFrameHandle** corresponding to **link**, which is interpreted as a control link; it should be either a PROCEDURE, POINTER TO FRAME or PROGRAM. If **link** is not a valid control link, then either the signal **InvalidGlobalFrame** or **UnboundProcedure** will be raised (see the section on Traps for a description of **UnboundProcedure**).

```
ValidateGlobalFrame: PROCEDURE [GlobalFrameHandle];
```

Checks to see that the parameter is a valid global frame; **InvalidGlobalFrame** is raised if not. Used to check the validity of **GlobalFrameHandle** parameters by system procedures such as **GlobalFrame**.

```
InvalidGlobalFrame: SIGNAL [frame: GlobalFrameHandle];
```

Indicates that **frame** does not point to a valid global frame.

```
EnumerateGlobalFrames: PROCEDURE [
    proc: PROCEDURE [GlobalFrameHandle] RETURNS [BOOLEAN]]
    RETURNS [GlobalFrameHandle];
```

Calls **proc** once for each global frame currently defined. If **proc** returns TRUE, **EnumerateGlobalFrames** returns the **GlobalFrameHandle** of the last frame processed. If all global frames have been processed, **NullGlobalFrame** is returned.

Warning: If new modules are created while **EnumerateGlobalFrames** is in control, it is not guaranteed that they will be included in the sequence of **GlobalFrameHandles** passed to **proc**.

Frames

A **Frame** is the implementation of the PROCEDURE language construct. All manipulation of **Frame**s is done using **FrameHandle**s.

FrameHandle: TYPE = POINTER TO **Frame**;

Frame: TYPE = RECORD [. . .];

Frames may be validated by calling:

ValidateFrame: PROCEDURE [**FrameHandle**];

Checks to see that the parameter is a valid frame; **InvalidFrame** is raised if not.

InvalidFrame: SIGNAL [**frame: FrameHandle**];

Indicates that **frame** does not point to a valid frame.

Module Creation / Deletion

The following procedures allow the programmer explicit control over the creation and deletion of modules. **NewConfig** and **RunConfig** provide an interface for loading configurations. The language construct NEW should be used for making copies of modules. **UnNew** and **UnNewConfig** allow the deletion of previously loaded modules and configurations. **SelfDestruct** allows a module to **UnNew** itself and return to its caller. These operations are defined in **FrameDefs**.

NewConfig: PROCEDURE [**name:** STRING];

Loads the BCD contained in the file **name**. A configuration loaded in this way can be started only by start traps or binding to and starting a frame or PROGRAM which it exports.

RunConfig: PROCEDURE [**name:** STRING];

Like **NewConfig**, except that the control module of the configuration will be started if there is one.

Warning: a stack error will result if the control modules requires parameters.

UnNewConfig: PROCEDURE [**frame: GlobalFrameHandle**];

Deletes all global frames that are a part of the configuration containing **frame**, as well as all copies of those frames. Frees the storage for the global frames (to the segmentation machinery if it was obtained from there, or to the frame heap), and frees all the code segments.

UnNew: PROCEDURE [**frame: GlobalFrameHandle**];

Deletes the global frame pointed to by **frame**. Returns the global frame to the frame heap (if the frame was allocated from that heap). Deletes the code segment of the frame if no other global frames share it.

Warning: there is no check for references that are bound to the module being deleted.

The language construct NEW and the operation **UnNew** should be used for creating and destroying copies of modules. They are relatively inexpensive operations, and provide facilities for using module instances as objects. **NewConfig** and **UnNewConfig** on the other hand, are relatively expensive operations, and should be used to add or remove configurations which are relatively static.

SelfDestruct: PROCEDURE;

UnNews the module of the caller, and returns to its caller's caller.

NoGlobalFrameSlots: SIGNAL;

Indicates that there is no room in the Global Frame Table when either **NewConfig** or **RunConfig** have been called or the lanuguage construct NEW has been invoked.

Fine point: global frames for single modules (and copies) are allocated from the frame heap. Other configurations allocate their global frames as a single (data) segment, and are subject to some wasted space due to breakage.

Other signals may be raised when **New**ing modules; these signals indicate that the BCD being loaded is invalid, versions of interfaces don't match, or code files cannot be found or compiled in non-Alto mode. In addition, when a module is started, the signal **StartFault** may be raised (see the section on **Traps** for more information).

Code Manipulation

The following procedures enable the user to control code swapping.

MakeCodeResident: PROCEDURE **[f: GlobalFrameHandle];**

Swaps in the code for **f** as low as possible in memory by pushing unlocked read-only segments out of the way. It will first swap out the code if it is swapped in.

LockCode: PROCEDURE **[link:** UNSPECIFIED**];**

Swaps in and locks the code segment associated with **link** (which may be either a PROCEDURE, POINTER TO FRAME or PROGRAM). The procedure **GlobalFrame** is used to find the global frame of **link** (and may raise the signals **InvalidGlobalFrame** and **UnboundProcedure**).

Warning: calling **LockCode** on a global frame that has not been started will disable start traps on that module.

UnlockCode: PROCEDURE **[link:** UNSPECIFIED**];**

Unlocks the code segment associated with **link** (which may be either a PROCEDURE, POINTER TO FRAME or PROGRAM). The procedure **GlobalFrame** is used to find the global frame of **link** (and may raise the signals **InvalidGlobalFrame** and **UnboundProcedure**).

SwapOutCode: PROCEDURE **[f: GlobalFrameHandle];**

Swaps out the code for **f**.

Fine point: when a frame has its code swapped out, all the global frames that have the same code segment will be updated to reflect the fact that the code is swapped out. These other frames either have their code segments packed with that frame or they are copies of that frame. Code may be swapped out by clients calling **SwapOutCode** or by the system.

SwapInCode: PROCEDURE **[f: GlobalFrameHandle];**

Swaps in and locks the code for **f**.

Warning: calling **SwapInCode** on a global frame that has not been started will disable start traps on that module.

Miscellaneous Operations

The following procedures are useful for debugging and determining what has been loaded.

GetCaller: PROCEDURE RETURNS **[PROGRAM];**

Returns the module containing its caller's caller.

IsBound: PROCEDURE **[link:** UNSPECIFIED**]** RETURNS **[BOOLEAN];**

In cases where configurations are conditionally loaded, **IsBound** can be used to determine the presence of a module or procedure. The UNSPECIFIED argument should be an imported procedure or program, or pointer to an imported variable.

Alto/Mesa Processes and Monitors

April 1979

This section has been taken from the *Pilot Functional Specification* and slightly modified to reflect the Alto/Mesa implementation.

Warning: The Alto/Mesa operating system software has not been revised and redesigned to fully exploit the capabilities of the new process mechanism. In particular, arbitrary preemptive processes are not supported, and the restrictions of Mesa 3.0 on processes running at interrupt level still apply.

Most aspects of processes and monitors are made available via constructs built into the Mesa language and described in Chapter 10 of the *Mesa Language Manual.* Some facilities whose frequency of use does not justify such treatment are cast as procedures, and form part of the Mesa system. The types described below are defined in **PSBDefs**.

PSB: PRIVATE TYPE = MACHINE DEPENDENT RECORD [. . .];

ProcessHandle: TYPE = POINTER TO **PSB**;

> A **PSB** defines the data structure underlying the Mesa process implementation. In general no client programs should be concerned with these types (except possibly for debugging).

The types and procedures described below are defined in **ProcessDefs**. Any of the operations which take a PROCESS as an argument (i. e., JOIN, **Abort**, and **Detach**) may generate the following signal.

InvalidProcess: SIGNAL **[process: ProcessHandle];**

> This signal indicates that the process argument does not correspond to any process known to the Mesa system. The check on the validity of the argument is *not* infallible. In particular, Mesa is unable to distinguish between a process which has been deleted and a new, recently created one which coincidentally has the same value.

A call to FORK may generate the following signal.

TooManyProcesses: ERROR;

> The Mesa system contains a fixed number of **PSB**s. This signal will result when there are no **PSB**s available to create a new process.

GetCurrent: PROCEDURE RETURNS **[ProcessHandle];**

> It returns the **ProcessHandle** of the current process.

Initialization

Every instance of a monitor and every condition variable must be initialized before it can be used. There are two cases:

> If the lock and the condition variables reside in the global frame (or possibly the local frame, in the case of the condition variables) Mesa will automatically initialize them along with the other global/local variables when the module is **STARTed** or the procedure is entered.
>
> If the lock and/or condition variables reside in client-allocated records, initialization is the responsibility of the client.
>
> Using uninitialized monitor locks or condition variables or reinitializing monitor locks or condition variables after they have have begun to be used will lead to totally unpredictable behavior.

The following operations are provided for initializing monitor locks and condition variables in client-created data structures.

InitializeMonitor: PROCEDURE [monitor: POINTER TO MONITORLOCK];

> **InitializeMonitor** leaves the monitor unlocked and sets the queue of waiting processes to empty. It may be called before or after the monitor data is initialized, but *must* be called before any entry procedure is invoked. Once use of the monitor has begun, **InitializeMonitor** *must never be called again.*

InitializeCondition: PROCEDURE [condition: POINTER TO CONDITION, ticks: Ticks];

> **InitializeCondition** sets the queue of waiting processes to empty and sets the timeout interval of the condition variable to **ticks** (measured in units of "ticks" of an internal clock). It may be called before or after the other monitor data is initialized, but *must* be called before any **WAIT** or **NOTIFY** operations are performed on the condition variable. Once use of the condition variable has begun, **InitializeCondition** *must never be called again.*

Clients may convert clock ticks, milliseconds and seconds using the following operations:

Ticks: TYPE = CARDINAL;

Milliseconds: TYPE = CARDINAL;

Seconds: TYPE = CARDINAL;

MsecToTicks: PROCEDURE [Milliseconds] RETURNS [Ticks];

TicksToMsec: PROCEDURE [Ticks] RETURNS [Milliseconds];

SecondsToTicks: PROCEDURE [Ticks] RETURNS [Seconds];

Timeouts

Condition variables which are initialized automatically are assigned a default timeout of a few seconds. The timeout of any condition variable may be changed by the following operation.

```
SetTimeout: PROCEDURE [condition: POINTER TO CONDITION, ticks: Ticks];

DisableTimeout: PROCEDURE [POINTER TO CONDITION];
```

SetTimeout adjusts the timeout interval for all subsequent **WAIT** operations applied to that condition variable. **DisableTimeout** disables timeouts for all subsequent **WAIT** operations applied to that condition variable. However, neither operation has any effect on processes which are already **WAIT**ing.

SetTimeout and **DisableTimeout** are the only available operations to adjust a condition variable once it has been used. In particular, **InitializeCondition** must not be used for this purpose, especially for condition variables which are automatically initialized.

Detaching Processes

A process which will never be joined is detached using the following operation.

```
Detach: PROCEDURE [PROCESS];
```

This operation sets the state of the process so that when it returns from its root procedure, it will be deleted immediately and its results, if any, will be discarded. If the process is invalid (e. g., if the process has already been deleted), the signal **InvalidProcess** may be generated.

The argument to **Detach** is actually of type **UNSPECIFIED**, and is validated as a **PROCESS** at run-time. This is necessary since there is no generic type which includes all **PROCESS** types, regardless of result types.

Priorities of Processes

When a process is created with **FORK**, it inherits the priority of the **FORK**ing process. If this proves unsatisfactory, the **FORK**ed process may change its own priority with the following operation.

```
SetPriority: PROCEDURE [Priority];

Priority: TYPE = [0..7];
```

A process may determine its own priority by calling:

```
GetPriority: PROCEDURE RETURNS [Priority];
```

There is no way for a process to alter the priority of another process.

CAUTION: *Use of multiple priorities in the Alto/Mesa implementation is severely restricted.* Any process running at other than the default priority (currently, 1) is forbidden to use many of the standard runtime support features of the Mesa environment. In practice, this means that non-standard priorities should be used only for interrupt handling, while all "normal" processing takes place concurrently at the default priority level. In addition, all interrupt level code must be locked in memory and should perform only a minimal amount of processing.

Aborting a process

A process can be aborted by calling the following operation.

Abort: PROCEDURE [PROCESS];

The effect of this operation is to generate the signal **Aborted** the next time the process executes a WAIT statement on *any* condition variable. If the process is already WAITing, an implicit NOTIFY is issued at the time this procedure is called.

The argument to **Abort** is actually of type **UNSPECIFIED**, and is validated as a **PROCESS** at run-time. This is necessary since there is no generic type which includes all **PROCESS** types, regardless of result types.

Aborted: ERROR;

The catch phrase for this signal may be attached to the WAIT statement, or it may be enabled in some scope according to the scope rules of Mesa. The catch phrase is executed with the corresponding monitor locked.

The intended use of **Abort** is to provide a means whereby one process may hint to another that the latter should go away, after first cleaning up. An **Abort** signal may occur on *any* condition variable, and thus *every* monitor should be protected by some catch phrase for it.

DisableAborts: PROCEDURE [POINTER TO CONDITION];

This procedure prevents a process from aborting when it waits on the specified condition variable. It is not implemented in Alto/Mesa.

Control of scheduling

The Mesa process mechanism does *not* attempt to allocate processor time fairly among processes of equal priority. Because of this, it may be desirable for a process which does not does not execute WAIT statements very frequently in the normal course of its computation to occasionally yield control of the processor by calling the following operation.

Yield: PROCEDURE;

This is a *hint* to the scheduler to run other processes of the same priority. However, *there is no guarantee that any other process will execute before the calling process resumes execution, even if there are processes able to execute.*

In no case must the logical correctness of client programs depend on the presence or absence of calls to **Yield**: priorities and yielding are *not* intended as a process-synchronization mechanism. They are only hints to assist clients in meeting performance requirements.

Interrupt Level Processes

This section refers only to the Alto/Mesa implementation. It should be of interest only to programmers of interrupt level code.

The Mesa monitor mechanism includes an extension to cover the case of communication between software processes and Input/Output controllers (hardware and/or firmware). This is done using the artifact of *naked condition variables*; that is, condition variables which are *not* effectively protected by a monitor lock. The need for this arises from the fact that communication with I/O controllers, while similar to normal interprocess communication, suffers from the problem that the controllers are intrinsically unable to enter monitors. This means that two important atomicity properties provided by monitor locks are lost:

> **Atomicity of wakeups:** Monitor locks eliminate the need for a traditional "wakeup waiting" (or "interrupt pending") flag. Lack of the monitor lock requires the provision of such a flag if lost interrupts are not to result.
>
> **Atomicity of data manipulations:** The monitor lock avoids the problems of critical races on shared data; traditionally, I/O architectures take an *ad hoc* approach to this problem, rather than providing any general mechanism.

The approach taken is to solve the first problem in a general way, and leave the second problem for case-by-case resolution by the designers of specific controller/software interfaces. (This closely mirrors the approach normally taken in more traditional I/O-interrupt architectures.)

A software process that deals with an I/O controller does so from within what appears to be a normal monitor. The monitor data includes the status and control blocks of the device and a condition variable which the device notifies to raise an "interrupt". Since the controller can access the shared data at any time, however, special care must be taken by the software to avoid conflicts. Similarly, if the controller tried to notify the software between the time it decided to wait on the condition variable, and the time that it actually performed the WAIT operation, the NOTIFY would be lost. To prevent this, the controller does a special form of NOTIFY (a *naked notify*), which sets a wakeup-waiting flag in the condition variable. This difference is invisible to the software, which does a normal WAIT operation on the condition variable.

The traditional operations **EnableInterrupts** and **DisableInterrupts** are provided for those rare circumstances in which seizing the entire machine is the only form of mutual exclusion which proves sufficient. Doing a WAIT while interrupts are disabled is not recommended.

```
InterruptLevel: TYPE = [0..15];
```

> **InterruptLevel**s correspond to the interrupt channels described in the *Alto Hardware Manual* (except that the numbering is different). Interrupt level 0 corresponds to Alto interrupt channel 15, the memory parity interrupt channel. Several levels are used by the Mesa system. Programmers of interrupt level code should see the definitions in **ProcessDefs**.

```
ConditionVector: TYPE = ARRAY InterruptLevel OF POINTER TO CONDITION;

CV: POINTER TO ConditionVector = -- magic constant --;
```

> **CV** points to an array of pointers to the naked condition variables described above. To associate a CONDITION with an interrupt level, assign a pointer to the CONDITION to the corresponding element of **CV** (**CV[level] ← @cVar**). To disable the naked NOTIFYs, assign NIL.

DisableInterrupts, EnableInterrupts: PROCEDURE;

These are **MACHINE CODE** procedures which disable and enable the handling of interrupts at the lowest level. They should be used only in matching pairs and only when no other exclusion mechanism will suffice. A counter is maintained of the number of unmatched **DisableInterrupts** so that nested pairs will work correctly.

Alto/Mesa Segment Package

April 1979

The Mesa virtual memory (VM) is organized as a vector of pages of size **PageSize** words; the last page is **MaxVMPage**. VM is occupied by segments: a segment is an integral number of pages in length and the words in a segment are all linearly addressable; segments have no empty holes in them. There are two variants of segments: *data* and *file*. *Data* segments are associated directly with memory and are not swappable or movable; *file* segments correspond to contiguous groups of pages in a file and may be swapped in and out of virtual memory.

Client programs access segments using **SegmentHandles**, which are pointers to **SegmentObjects**. A **SegmentObject** contains the information necessary to describe the segment. Although there are some routines that deal with **SegmentHandles**, segments are not particulatily interesting unless they are discriminated as to data or file variant.

Client programs access file segments using **FileSegmentHandles**, which are pointers to **file SegmentObjects**. A file segment contains sufficient information to compute its address if the segment is swapped in. Internally, the segmentation package maintains a set of objects called **FileObjects**: a file object, among other things, contains the file's disk address and serial number, as well as its access rights. The association between a segment and a file is made when the segment is created. The Mesa file package is documented separately.

Segments may also be pages in VM rather than attached to a file. Such data segments are not swappable or movable in any way (relative to the Mesa virtual memory). Thus, absolute pointers into a data segment are valid for the lifetime of the segment. **DataSegmentHandles** and **data SegmentObjects** are used to record information about these segments. See **SegmentDefs** for further details.

Segments

As mentioned above, segment objects come in two varieties: data segments and file segments The following structure contains the necessary information:

```
SegmentHandle: TYPE = POINTER TO SegmentObject;

SegmentType: TYPE = {data, file};

SegmentObject: TYPE = RECORD [
   . . . ,
   SELECT type: SegmentType FROM
      data => [
         VMpage: [0..MaxVMPage],          -- VM page number
         pages: [1..MaxVMPage + 1]        -- number of pages
         . . .],
      file => [
         swappedin: BOOLEAN,              -- TRUE iff segment is in
         read, write: BOOLEAN,            -- access options
```

```
    class: FileSegmentClass,        -- {code, other}
    VMpage: [0..MaxVMPage],  -- if swapped in, VM page number
    file: FileHandle,               -- see the file package
    inuse: BOOLEAN,                 -- software LRU bit
    base: PageNumber,               -- first page of the file to include
    pages: [1..MaxVMPage + 1],      -- number of pages, beginning with base
    lock: [0..MaxLocks],            -- locking reference count
     . . .],
ENDCASE];
```

The procedures which manipulate undiscriminated segments are:

VMtoSegment: PROCEDURE [a: POINTER] RETURNS [SegmentHandle];

The handle for the segment containing the specified address (as currently laid out in memory) is returned; NIL is returned if no segment contains it. This does not imply that the page containing the address is free, however; it may be reserved for some operation currently in progress.

SegmentAddress: PROCEDURE [seg: SegmentHandle] RETURNS [POINTER];

The address of the beginning of the segment is returned; NIL is returned if the segment is a file segment and not currently swapped in. To guarantee the validity of the address the segment should be locked when this procedure is called (see below), since the system may swap out file segments which are not locked. *Beware of dangling references!*

Data Segments

Data segments are associated only with virtual memory (there is no swapping file), and are never moved or swapped out.

DataSegmentHandle: TYPE = POINTER TO DataSegmentObject;

DataSegmentObject: TYPE = data SegmentObject;

The procedures which manipulate data segments are:

NewDataSegment: PROCEDURE [base: PageNumber, pages: PageCount]
RETURNS [DataSegmentHandle];

Creates a new data segment and returns a handle for it. If **base** is **DefaultBase** then the segment is allowed to begin on any free page in memory. If **base** is an actual page number (in **[0..MaxVMPage]**), an attempt is made to place the segment at that location. Note that **pages** should not be defaulted.

DataSegmentAddress: PROCEDURE [seg: DataSegmentHandle] RETURNS [POINTER];

Returns a pointer to the base of the segment in virtual memory. In the current implementation, segments always begin on a page boundary; this may not be true in the (distant) future.

```
VMtoDataSegment: PROCEDURE [a: POINTER] RETURNS [DataSegmentHandle];
```

The handle for the segment containing the specified address (as currently laid out in memory) is returned. NIL is returned if no data segment contains it. This does not imply that the page containing the address is free, however; it may be assigned to a file segment, or reserved for some operation currently in progress.

```
DeleteDataSegment: PROCEDURE [seg: DataSegmentHandle];
```

The specified data segment is deleted and its segment object freed. When a segment is successfully deleted, any VM which it occupied becomes free.

```
EnumerateDataSegments: PROCEDURE [
    proc: PROCEDURE [DataSegmentHandle] RETURNS [BOOLEAN]]
    RETURNS [DataSegmentHandle];
```

Calls **proc** once for each data segment currently defined. If **proc** returns TRUE, **EnumerateDataSegments** returns the handle of the last segment processed. If the end of the set of data segments is reached, NIL is returned.

If data segments are created while **EnumerateDataSegments** is in control, it is not guaranteed that they will be included in the sequence of **DataSegmentHandles** passed to **proc**.

File Segments

Unlike data segments, file segments are associated with a contiguous group of pages in a file and are therefore swappable. Pointers into a file segment are valid only while it is swapped in (and locked so that it will not be swapped out). A file segment which is swapped out occupies no space in virtual memory, other than the segment object which describes it. If the LRU bit (**inuse**) is set, the swapper will ignore this segment on its first search for free pages. The swapper resets this bit, so the segment will be considered if a second pass is necessary.

```
FileSegmentHandle: TYPE = POINTER TO FileSegmentObject;

FileSegmentObject: TYPE = file SegmentObject;
```

To create new file segments, use

```
NewFileSegment: PROCEDURE [
    file: FileHandle, base: PageNumber, pages: PageCount, access: AccessOptions]
    RETURNS [FileSegmentHandle];
```

Creates a new segment and returns a handle for it. The segment is associated with the corresponding file pages, but the file is not opened and the segment is not swapped in. If **base** is **DefaultBase**, the segment will begin with the first data page of the file, and if **pages** is **DefaultPages**, it will include the last page of the file. Although it is generally not done, a segment can begin with the leader page (page zero) of a file. Finally, if **access** is **DefaultAccess**, read access is assumed.

If the access specifies that changing the data is permitted, then whenever it is necessary to swap this segment out and remove its pages from memory, pages will be written to the file (the Alto has no hardware to detect if the pages have actually been changed). Note that it is possible to change the

segment's access (by setting the **write** bit, for example), provided the file to which it is attached has the appropriate access rights.

FileSegmentAddress: PROCEDURE **[seg: FileSegmentHandle]** RETURNS **[**POINTER**];**

> The address of the beginning of the segment is returned. The signal **SwapError** is raised if the segment is not currently swapped in. To guarantee the validity of the address, the segment should be locked when this procedure is called (see below), since the system may swap out file segments which are not locked. *Beware of dangling references!*

VMtoFileSegment: PROCEDURE **[a:** POINTER**]** RETURNS **[FileSegmentHandle];**

> The handle of the file segment containing the specified address (as currently laid out in memory) is returned; NIL is returned if no file segment contains it. This does not imply that the page containing the address is free, however; it may be assigned to a data segment, or reserved for a segment currently being swapped in.

DeleteFileSegment: PROCEDURE **[seg: FileSegmentHandle];**

> The specified file segment is deleted and its segment object is released. If the segment is swapped in, it is first swapped out (it should not be locked). If there are no other segments associated with this segment's file (the file's **segcount** is zero), then **ReleaseFile** is called to release the **FileObject**. When a segment is successfully deleted, any VM which it may have occupied becomes free.

EnumerateFileSegments: PROCEDURE **[**
proc: PROCEDURE **[FileSegmentHandle]** RETURNS **[**BOOLEAN**]]**
RETURNS **[FileSegmentHandle];**

> Calls **proc** once for each data segment currently defined. If **proc** returns TRUE, **EnumerateFileSegments** returns the handle of the last segment processed. If the end of the set of data segments is reached, NIL is returned.

If file segments are created while **EnumerateFileSegments** is in control, it is not guaranteed that they will be included in the sequence of **FileSegmentHandles** passed to **proc**.

Window Segments

A window segment is similar to a file segment, except that the **base** and **pages** fields of the segment may be altered (using the procedures described below) after it is created, in order to slide the window around in a file or to vary the window's size. In reality, all file segments are in fact window segments, and may be moved with the following procedure:

MoveFileSegment: PROCEDURE **[**
seg: FileSegmentHandle, base: PageNumber, pages: PageCount];

> If the segment is swapped in, it is first swapped out (it should not be locked). The segment is then moved to the new location in the segment's file, but it is not swapped in. The **base** and **pages** are defaulted as in **NewFileSegment**.
>
> Fine point: in the current implementation, the disk address of the original segment is retained as a hint about the new location, thus improving performance considerably when a one page segment is slid forward or backward in a file.

If the original and final position of the segment overlap, there is no guarantee that the overlapping pages are actually written, nor is it guaranteed that a minimum number of pages are transferred. The segment package reserves the option to implement (or not to implement) such optimizations in the future.

Swapping Segments

A segment can be swapped into and out of VM. The procedures and signals which implement this are described in this section:

SwapIn: PROCEDURE [seg: FileSegmentHandle];

> Swaps in the specified segment (if it is swapped out), opening the associated file if necessary; locks it so it won't be moved or swapped out. A **SwapError** will result if the segment already has **MaxLocks** locks on it, or if the segment's file has **MaxRefs** segments currently attached to it and swapped in.

To unlock a segment (allow it to be swapped), use the procedure

Unlock: PROCEDURE [seg: FileSegmentHandle];

> Unlocks the specifed segment so that it can be swapped out. Note that locking behaves like reference counting, so that locks (performed by **SwapIn**) must be properly paired with **Unlocks**.

A segment is swapped out using

SwapOut: PROCEDURE [seg: FileSegmentHandle];

> Swaps out the specified segment, writing the pages back to the file if the segment's access makes this necessary, and free the segment's VM pages. If the segment is locked, a **SwapError** will be generated.

A program may explicitly request that the file pages corresponding to a segment be updated by calling:

SwapUp: PROCEDURE [seg: FileSegmentHandle];

> Write the pages of the segment back to the file if the access requires it. This operation does not unlock the segment or free the segment's VM pages.

Note that neither **SwapIn**, **SwapOut**, or **SwapUp** are capable of extending a file (physically adding pages or bytes to it) based on the size of a segment. Segments may be attached only to pages of a file that are already allocated on the disk (and chained together). Extending (or contracting) a file must be done using other mechanisms (for example, see **SetEndOfFile** in the file package).

Signals

The following signals may be generated by the segment package:

InvalidSegmentSize: SIGNAL **[pages: PageCount];**

In **NewDataSegment** or **NewFileSegment** a zero length segment has been requested, or the length exceeds the size of virtual memory.

InsufficientVM: SIGNAL **[needed: PageCount];**

In **NewDataSegment** or **SwapIn** there is not enough contiguous memory to accomodate a segment; **needed** is the number of pages that are actually required. If resumed, the allocation will be retryed; this gives the catcher of this signal a chance to free up some VM. Users can free VM pages by deleting data segments and by allowing locked segments to become swappable (see also the section below on swapping strategies).

VMnotFree: SIGNAL **[base: PageNumber, pages: PageCount];**

In **NewDataSegment** the base was not **DefaultBase** and the specified memory pages were not free.

SwapError: SIGNAL **[seg: FileSegmentHandle];**

An invalid swapping operation was attempted with **seg**.

SegmentFault: SIGNAL **[seg: FileSegmentHandle, pages: PageCount];**

End of file was encountered while attempting to swap the segment in or out; **pages** is the actual number of pages in the segment. If **pages** is greater than zero then the signal may be resumed; the segment will be truncated accordingly (of course, this will not alter the file length).

Low Level Memory Allocation

Operations are provided for users that have a need to control memory allocation at a lower level than provided above. Allocation is controlled by the information in an **AllocInfo** (which is passed along with each operation).

```
AllocInfo: TYPE = RECORD [
  . . .
  effort: {hard, easy},
  direction: {topdown, bottomup},
  . . . ];
```

If the **effort** field is **hard**, unlocked read-only file segments will be pushed out of the way. The **direction** field specifies the direction of the search for a hole. In the above procedures, data segments are allocated **topdown**, and file segments are allocated **bottomup**. See **AllocDefs** for further information.

The operations which form the low-level memory allocation are:

```
MakeDataSegment: PROCEDURE [
    base: PageNumber, pages: PageCount, info: AllocInfo]
    RETURNS [DataSegmentHandle];
```

Acts like **NewDataSegment**, except **info** is passed as an additional parameter; the same restrictions apply and the same signals may be generated.

```
MakeSwappedIn: PROCEDURE [
    seg: FileSegmentHandle, base: PageNumber, info: AllocInfo];
```

Acts like **SwapIn**, except **info** and **base** are passed as additional parameters; the same restrictions apply and the same signals may be generated. If **base** is **DefaultBase** then the segment is allowed to begin on any free page in memory. If **base** is an actual page number (in **[0..MaxVMPage]**), an attempt is made to place the segment at that location (**VMnotFree** will be raised if the specified pages are not available).

Swapping Strategies

A mechanism is provided for informing the segmentation package of emergency measures which can be taken when the signal **InsufficientVM** is (about to be) generated. These measures take the form of **SwappingProcedures** which, when called by the swapping manager, attempt to make more room in virtual memory and return a BOOLEAN indicating their success or failure to do so. The swapping manager invokes each procedure in turn, retrying the allocation after each procedure which has indicated success, until sufficient memory is obtained. If all such procedures indicate failure, the signal **InsufficientVM** is raised (the swapping manager is not crying wolf!).

The swapping strategies are maintained as a linked list of **SwapStrategy** nodes whose procedures are invoked from head to tail.

```
SwappingProcedure: TYPE = PROCEDURE [
    needed: PageCount, info: AllocInfo, seg: SegmentHandle]
    RETURNS [BOOLEAN];
```

The following parameters are supplied to swapping procedures to allow them the make intelligent decisions about making room: **needed** is the number of pages requested, **info** the **AllocInfo** supplied to the allocator, and **seg** the file segment that will use the allocated memory (NIL if the segment is not known).

```
SwapStrategy: TYPE = RECORD [
    link: POINTER TO SwapStrategy,
    proc: SwappingProcedure];
```

The swapping manager initializes the list with a single node which invokes code swapping as a last resort.

```
StrategyList: POINTER TO SwapStrategy ← @LastResort;

LastResort: SwapStrategy = SwapStrategy[NIL, TryCodeSwapping].
```

Swapping procedures are added to and removed from the list by the procedures:

```
AddSwapStrategy: PROCEDURE [strategy: POINTER TO SwapStrategy];
```

The specified strategy node **strategy** is added to the head of the list of swapping procedures. If **strategy** is already on the list, its position and content are not disturbed.

```
RemoveSwapStrategy: PROCEDURE [strategy: POINTER TO SwapStrategy];
```

The specified strategy node s is removed from the list of swapping procedures.

Currently, **TryCodeSwapping** uses an (approximately) LRU (least-recently-used) algorithm to choose a code segment to swap out. Only code segments which are not locked are considered. Unlocked read-only file segments are also swapped out by **TryCodeSwapping**.

Since it is unattractive to require that swapping strategies (other than **TryCodeSwapping**) be locked, swapping procedures should observe the following conventions. If such a procedure obtains a state in which it has nothing to swap, it should either remove the node containing it from the strategy list or change the procedure in the node to be

```
CantSwap: SwappingProcedure;
```

Because **CantSwap** is part of the swapping manager (and therefore locked), this will avoid swapping in a strategy procedure which knows it has nothing to do.

Miscellaneous Procedures

The following procedures implement conversion between memory addresses and virtual memory page numbers.

```
PageFromAddress: PROCEDURE [a: POINTER] RETURNS [PageNumber];

AddressFromPage: PROCEDURE [p: PageNumber] RETURNS [POINTER];

PagePointer: PROCEDURE [a: POINTER] RETURNS [POINTER];
```

PagePointer returns the address of the beginning of the page which contains its argument.

The following procedures implement conversion between **FileSegment**s and **DataSegment**s. Note that both segments must exist at the time of the call, and neither is destroyed. They must be of the same length (a **SwapError** will result otherwise).

```
CopyDataToFileSegment: PROCEDURE [
   dataseg: DataSegmentHandle, fileseg: FileSegmentHandle];
```

initializes a file segment to be the contents of a data segment.

```
CopyFileToDataSegment: PROCEDURE [
   fileseg: FileSegmentHandle, dataseg: DataSegmentHandle];
```

initializes a data segment to be the contents of a file segment.

Alto/Mesa Storage Management Facilities

April 1979

Two collections of Mesa procedures are available for acquiring and managing storage areas. The *segmentation machinery*, which is described in detail elsewhere, provides contiguous groups of *pages* (256 word blocks) in the virtual memory. A simplified interface with that machinery is described below. There is also a Mesa *free storage package* for managing arbitrarily sized *nodes* within free storage *zones*. Since all state information is recorded within the zones themselves, the system-provided instantiation of the latter package can manage an arbitrary number of zones. There is one system-defined zone, called the *free storage heap*, available for general use; special procedures exist for creating and destroying nodes within it. The salient characteristics of these packages are summarized below.

The segmentation machinery is most suitable for obtaining large blocks of storage. All bookkeeping information associated with such blocks is recorded in auxiliary tables that are managed by the segmentation system, not in the blocks themselves. Allocating or releasing a segment involves searching and updating a number of those tables. On the other hand, any freed page becomes available for general use by the system (loading, buffering, etc.) and any two adjacent free pages can be coalesced to become part of a new segment.

The free storage package is a transliteration of a BCPL program by Ed McCreight that was itself based upon a suggestion by Don Knuth (*The Art of Computer Programming*, Volume 1, p. 453, #19). Within a zone, free nodes are kept as a linked list. One hidden word containing bookkeeping information is stored with each allocated node, and additional bookkeeping information is kept in the header of each zone. Allocation and release of nodes are usually very fast. Adjacent free nodes are always able to be coalesced. It is also possible to add new areas of storage to enlarge a zone. These new areas are linked together so that they may be deleted if all the nodes in an area are free; in addition, an entire zone may be deleted.

The free storage package performs best when the sizes of nodes are small compared to the sizes of the block(s) making up the zone. In particular, the system's heap is intended to be used for small, transient data structures, such as the nodes of a temporary list structure or the bodies of (short) strings when the maximum length must be computed dynamically or the structure must outlive the frame that creates it. Use of the heap for large (i.e., multipage) nodes decreases flexibility in storage management, since the additional pages may become a permanent part of the zone.

The allocators in both packages return absolute pointers; allocated nodes are not relocatable and there is no garbage collection or automatic deallocation of any sort. Also, the values returned by the allocators are free pointers (type POINTER TO UNSPECIFIED) which must be cast appropriately (usually by assignment) before they can be used.

Segmentation Interface

The following definitions are contained in **SystemDefs**.

AllocateSegment: PROCEDURE [**nwords:** CARDINAL] RETURNS [**base:** POINTER];

> Allocates a segment of virtual memory containing at least **nwords** words and returns the address of the first word in that segment. **AllocateSegment** provides a simple interface to **NewDataSegment** for allocating VM segments only; see the description of that procedure for further explanation.

AllocateResidentSegment: PROCEDURE [**nwords:** CARDINAL] RETURNS [**base:** POINTER];

> Behaves like **AllocateSegment**, except that unlocked read-only file segments are pushed out of the way when the segment is allocated.

SegmentSize: PROCEDURE [**base:** POINTER] RETURNS [**nwords:** CARDINAL];

> Returns the number of words actually obtained in the segment.

These procedures allow complete utilization of segments obtained without knowledge of page structure and guaranteed only to have some minimum size. Such segments are returned to the system by

FreeSegment: PROCEDURE [**base:** POINTER];

For uses in which the page structure is already known, the following procedures are also provided.

AllocatePages: PROCEDURE [**npages:** CARDINAL] RETURNS [**base:** POINTER];

AllocateResidentPages: PROCEDURE [**npages:** CARDINAL] RETURNS [**base:** POINTER];

PagesForWords: PROCEDURE [**nwords:** CARDINAL] RETURNS [**npages:** CARDINAL];

FreePages: PROCEDURE [**base:** POINTER];

Any storage obtained using **AllocatePages** or **AllocateResidentPages** is guaranteed to begin on a page boundary.

Free Storage Package

The following definitions are available in **FSPDefs**. A *zone* is a block of storage containing embedded *nodes*. The length of either a zone or a node is

BlockSize: TYPE = INTEGER [0..VMLimit/2]; -- *15 bits.*

Each zone is headed by a **ZoneHeader**, which is a monitored record with the following public fields:

```
. . .
threshold: BlockSize,  -- minimum node size in zone
checking: BOOLEAN,  -- zone checking (see below)
. . .
```

Zones are identified by pointers of type

```
ZonePointer: TYPE = POINTER TO ZoneHeader;
```

Associated with each zone is a procedure of type

```
Deallocator: TYPE = PROCEDURE [POINTER];
```

which is used to deallocate the storage used by the zone. The following **Deallocator** may be supplied when nothing is to be done to the storage being freed:

```
DoNothingDeallocate: Deallocator;
```

Zone Operations

An arbitrary block of (uninterpreted) storage is converted to a zone by the procedure

```
MakeNewZone: PROCEDURE [
   base: POINTER, length: BlockSize, deallocate: Deallocator]
   RETURNS [z: ZonePointer];
```

Such a block can alternatively be made an extension of an existing zone by calling

```
AddToNewZone: PROCEDURE [
   z: ZonePointer, base: POINTER, length: BlockSize, deallocate: Deallocator];
```

The following procedures default **DoNothingDeallocate** as the **Deallocator**:

```
MakeZone: PROCEDURE [base: POINTER, length: BlockSize] RETURNS [z: ZonePointer];

AddToZone: PROCEDURE [z: ZonePointer, base: POINTER, length: BlockSize];
```

Unused areas of the zone are released by calling

```
PruneZone: PROCEDURE [z: ZonePointer] RETURNS [BOOLEAN];
```

which returns **TRUE** if any areas were freed and **FALSE** otherwise.

A zone may be destroyed and all its storage freed by calling

```
DestroyZone: PROCEDURE [z: ZonePointer];
```

No check is made for any nodes that are in use.

Warning: This operation cannot be protected by the monitor, and can therefore result in severe errors if another process is inside the zone.

Node Operations

The largest node that can be allocated in a virgin block of size **length** is **length-ZoneOverhead**.

A node is allocated by

```
MakeNode: PROCEDURE [z: ZonePointer, n: BlockSize] RETURNS [POINTER];
```

The value returned points to a block of **n** words; there is an additional hidden word of overhead (at offset-1) which must be preserved by users of the node. Nodes are sometimes split to satisfy allocation requests. Splitting within a zone **z** never generates fragments with size less than **z.threshold**, which is initialized to the minimum size of a free node. A request for a node of size **n** will produce a node with size in the range **[n . . n + z.threshold)**.

The actual size of an allocated node is returned by

```
NodeSize: PROCEDURE [p: POINTER] RETURNS [BlockSize];
```

If after coalescing all free nodes, a node of the requested size cannot be found,

```
NoRoomInZone: SIGNAL [z: ZonePointer];
```

is raised. This signal can be resumed (after, e.g., adding to the zone), and another attempt to allocate and return a suitable node will be made. An allocated node is returned to the zone by

```
FreeNode: PROCEDURE [z: ZonePointer, p: POINTER];
```

Alternatively, an existing node can be split by calling

```
SplitNode: PROCEDURE [z: ZonePointer, p: POINTER, n: BlockSize];
```

the first **n** words of the node **p** remain allocated, and the remainder of the node is freed.

When a zone **z** is created, the variable **z.checking** is initialized to FALSE. If that variable is set to TRUE, the zone is checked for consistency prior to each transaction involving that zone. A failure raises one of the following signals:

```
InvalidZone, InvalidNode: ERROR [POINTER];
```

Allocation From The Heap

Clients who use heap storage extensively are encouraged to create their own heap from the above operations. The following procedures provide a simple interface to the free storage package.

```
myHeap: FSPDefs.ZonePointer ← NIL;

GetSpace: PROCEDURE [nwords: CARDINAL] RETURNS [p: POINTER] =
   BEGIN OPEN SystemDefs, FSPDefs;
   np: CARDINAL;
   p ← MakeNode[myHeap, nwords !
      NoRoomInZone =>
         BEGIN
         np ← PagesForWords[nwords + ZoneOverhead + NodeOverhead];
         AddToNewZone[
            myHeap, AllocateResidentPages[np], np*AltoDefs.PageSize, FreePages];
         RESUME
         END];
   RETURN
   END;
```

```
FreeSpace: PROCEDURE [p: POINTER] =
   BEGIN
   FSPDefs.FreeNode[myHeap, p];
   RETURN
   END;

GetString: PROCEDURE [nchars: CARDINAL] RETURNS [s: STRING] =
   BEGIN
   s ← GetSpace[StringDefs.WordsForString[nchars]];
   st ← [length: 0, maxlength: nchars, text:];
   RETURN
   END;

FreeString: PROCEDURE [s: STRING] = LOOPHOLE[FreeSpace];

InitHeap: PROCEDURE [npages: CARDINAL] =
   BEGIN OPEN SystemDefs, FSPDefs;
   IF myHeap # NIL THEN EraseHeap[];
   myHeap ← MakeNewZone[
      AllocateResidentPages[npages], npages*AltoDefs.PageSize, FreePages];
   RETURN
   END;

EraseHeap: PROCEDURE =
   BEGIN
   FSPDefs.DestroyZone[myHeap];
   myHeap ← NIL;
   RETURN
   END;
```

Clients who use the heap infrequently may use the system storage heap. The following definitions are available in **SystemDefs**. The heap is managed by the free storage package; the appropriate zone pointer for use with the procedures described in the previous section is returned by

```
HeapZone: PROCEDURE RETURNS [ZonePointer];
```

The following procedures provide a specialized interface.

```
AllocateHeapNode: PROCEDURE [nwords: CARDINAL] RETURNS [p: POINTER];

FreeHeapNode: PROCEDURE [p: POINTER];
```

In addition,

```
AllocateHeapString: PROCEDURE [nchars: CARDINAL] RETURNS [s: STRING];
```

> allocates space for the body of a string in the heap. The field **s.length** is set to zero; **s.maxlength** is set to **nchars**.

Such strings are freed by

```
FreeHeapString: PROCEDURE [s: STRING];
```

If an allocation request cannot be satisfied from existing heap storage, an attempt is made to extend the heap with a block of appropriate size obtained from the segmentation machinery. The extension becomes a permanent part of the heap.

The heap may be pruned by calling

```
PruneHeap: PROCEDURE RETURNS [BOOLEAN];
```

which returns TRUE if any storage was returned to the segmentation machinery, and FALSE otherwise.

Alto/Mesa StreamIO Package

April 1979

StreamIO contains a set of procedures for convenient use of the character and string stream facilities in Mesa. The procedures of the **StreamIO** package are described below. The declarations necessary to use the procedures are in **IODefs**.

StreamIO uses two streams, one for input and one for output. These may be gotten by calling:

GetInputStream, GetOutputStream: PROCEDURE RETURNS **[StreamHandle];**

Returns the stream used for input or output, respectively.

The streams may be changed by calling:

SetInputStream, SetOutputStream: PROCEDURE **[StreamHandle];**

Replaces the input or output stream, respectively.

Character IO

ReadChar: PROCEDURE RETURNS **[**CHARACTER**];**

Returns the next character from the **InputStream.**

WriteChar: PROCEDURE **[c:** CHARACTER**];**

The character **c** is written on the **OutputStream.**

Definitions for control characters such as **NUL, BS, TAB, LF, FF, CR, ESC, SP, DEL,** and **ControlA - ControlZ** can be found in **IODefs.**

String Input

The procedures below read input from the **InputStream.** The following exceptional conditions may occur.

LineOverflow: SIGNAL **[s:** STRING**]** RETURNS **[ns:** STRING**];**

The input has filled the string **s**, the current contents of the string is passed as a parameter to the SIGNAL. The catch phrase should return a string **ns** with more room.

Rubout: SIGNAL;

The DEL key was typed during **ReadEditedString.**

The **SetEcho** procedure controls the echoing of characters on input. It returns the previous state of the echoing mode which is defaulted to on.

SetEcho: PROCEDURE [**new:** BOOLEAN] RETURNS [**old:** BOOLEAN];

The input procedures are:

ReadEditedString: PROCEDURE [
s: STRING, **t:** PROCEDURE [CHARACTER] RETURNS [BOOLEAN], **newstring:** BOOLEAN]
RETURNS [CHARACTER];

s contains (on return) the string read from the **InputStream.** The procedure **t** should return TRUE if the CHARACTER passed to it should terminate the string. If (**newstring** is TRUE and the first input character is ESC) or (**newstring** is FALSE), then **s** is treated as if it had been read from **InputStream** (input characters are appended to it). Otherwise **s** is initialized to be empty before reading is begun.

A string is read from the **InputStream** with the following editing characters recognized:

↑A, ↑H (BS) delete the last character
↑W, ↑Q delete the last word
↑X delete the line and start over
↑R retype the line
↑V quote the next character

If echoing is on all characters except the terminating character are echoed on the **OutputStream.** The user supplied procedure **t** determines which character(s) terminate the string. The character returned is the character which terminated the string and is not echoed or included in the string.

The following procedures all call **ReadEditedString** passing TRUE for **newstring.**

ReadString: PROCEDURE [**s:** STRING, **t:** PROCEDURE [CHARACTER] RETURNS [BOOLEAN]];

Like **ReadEditedString** except that the terminating character is echoed. No value is returned.

ReadLine: PROCEDURE [**s:** STRING];

Reads from the **InputStream** up to the next carriage return character using **ReadEditedString.** The terminating character is not part of **s.**

ReadID: PROCEDURE [**s:** STRING];

Uses **ReadEditedString** to read a string terminated with a space or carriage return into **s.** The terminating character is not echoed.

String Output

WriteString: PROCEDURE [**s:** STRING];

The string **s** is written on the **OutputStream.**

WriteLine: PROCEDURE [**s:** STRING];

The string **s** is written on the **OutputStream** followed by a carriage return.

Number Input

These procedures use the **StringToNumber** conversion procedures from the **Strings** package.

ReadNumber: PROCEDURE [**default:** UNSPECIFIED, **radix:** CARDINAL]
RETURNS [UNSPECIFIED];

ReadID followed by **StringToNumber.** The value **default** will be displayed if ESC is typed. **radix** is a default value, use the "**B**" or "**D**" notation to force octal or decimal. **radix** values other than 8 or 10 cause unpredictable results.

ReadDecimal: PROCEDURE RETURNS [INTEGER];

ReadID followed by **StringToDecimal.**

ReadOctal: PROCEDURE RETURNS [UNSPECIFIED];

ReadID followed by **StringToOctal.**

Number Output

NumberFormat: TYPE = RECORD [
base: [2..36], zerofill, unsigned: BOOLEAN, **columns: [0..255]];**

refers to a number whose base is **f.base**; the field is **f.columns** wide; if **f.zerofill**, the extra columns are filled with zeros, otherwise spaces are used; if **f.unsigned**, the number is treated as unsigned.

OutNumber: PROCEDURE [**StreamHandle,** UNSPECIFIED, **NumberFormat];**

Converts the value to a character string of digits as specified by the **NumberFormat** and outputs them to the **StreamHandle.**

WriteNumber: PROCEDURE [UNSPECIFIED, **NumberFormat];**

Equivalent to **OutNumber** with **OutputStream** as the **StreamHandle.**

WriteDecimal: PROCEDURE [n: INTEGER];

The value of **n** is converted to a character string of digits in base ten and output to the **OutputStream**. Negative numbers are written with a preceeding minus sign ('-).

WriteOctal: PROCEDURE [n: UNSPECIFIED];

The value of **n** is converted to a character string of digits in base eight and output to the **OutputStream**. The numbers are unsigned, i.e., -2 is written as **177776B**. The "**B**" is appended to any number more than one digit long.

Initialization

The Mesa system provides an instance of **StreamIO** which will obtain input from the keyboard and write output to the display. Client programs may create new instances of **StreamIO** to deal with other streams by writing:

```
StreamIO: FROM "streamio";
        . . .
  IMPORTS  . . . systemio: StreamIO . . . ;
        . . .
  f: POINTER TO FRAME [StreamIO];
        . . .
  f ← NEW systemio;
  START f;
```

The streams used for input and output can then be set by calling **SetInputStream** or **SetOutputStream**. The desired stream procedures may be accessed by OPENing **f** or writing **f.procedurename**.

Alto/Mesa Streams

April 1979

Streams provide a standard interface between programs and their sources of sequential input and their sinks for sequential output. A set of standard operations defined for all types of streams is sufficient for all ordinary input-output requirements. In addition, most streams have special (device dependent) operations defined for them; programs which use such operations thereby forfeit complete compatibility.

Streams transmit information in atomic units called items. Usually an item is a **CHARACTER** or a **WORD**, and this is the case for most of the streams supplied with Mesa. Of course, a stream supplied to a program must have the same ideas about the kind of item it handles as the program does; otherwise confusion will result. Normally, streams which transmit text use **CHARACTER** items, and those which transmit binary information use **WORD**s.

Streams are passed about using **StreamHandle**s, variants of which are produced by the (device dependent) procedures that create streams. A **StreamHandle** is a pointer to a variant record of type **StreamObject**, which is defined (in **StreamDefs**) as follows:

```
StreamHandle: TYPE = POINTER TO StreamObject;

KeyboardHandle: TYPE = POINTER TO Keyboard StreamObject;
DisplayHandle: TYPE = POINTER TO Display StreamObject;
DiskHandle: TYPE = POINTER TO Disk StreamObject;

StreamObject: TYPE = RECORD [
    reset: PROCEDURE [StreamHandle],
    get: PROCEDURE [StreamHandle] RETURNS [UNSPECIFIED],
    putback: PROCEDURE [StreamHandle, UNSPECIFIED],
    put: PROCEDURE [StreamHandle, UNSPECIFIED],
    endof: PROCEDURE [StreamHandle] RETURNS [BOOLEAN],
    destroy: PROCEDURE [StreamHandle],
    link: StreamHandle,
    body: SELECT type: * FROM
      Keyboard => . . .
      Display => . . .
      Disk => . . .
      Other => [type: UNSPECIFIED, data: POINTER];
```

The procedures which create streams return descriminated pointers (a **DiskHandle**, for example), which can be assigned to variables of type **StreamHandle** without any loopholes. Most stream procedures (and all of the standard operations) expect **StreamHandle**s (which can be matched by any descriminated pointer); they check at runtime for the appropriate stream type.

Error conditions are reported in a fashion independent of the particular stream type, using the following definitions (not all error codes are applicable to all stream types):

```
StreamError: SIGNAL [stream: StreamHandle, error: StreamErrorCode];

StreamErrorCode: TYPE = {
    StreamType, StreamAccess, StreamOperation,
    StreamPosition, StreamEnd, StreamBug};
```

As the definition implies, each stream object contains procedures that implement the standard stream operations, as described below (**s** is a **StreamHandle**, **i** is an item of the appropriate type, and "**code** error" means that SIGNAL **StreamError[s, code]** is raised):

> **reset[s]** restores the stream to some initial state, generally as close as possible to the state it is in just after it is created.
>
> **get[s]** returns the next item; **StreamAccess** error if **s** cannot be read or if **endof[s]** is true before the call.
>
> **putback[s, i]** modifies the stream so that the next **get[s]** will return **i** and leave **s** in the state it was in before the **putback**.
>
> **put[s, i]** writes **i** into the stream as the next item; **StreamAccess** error if the stream cannot be written; **StreamEnd** error if there is no more space in the stream.
>
> **endof[s]** TRUE if there are no more items to be gotten from **s**. For output streams, **endof** is device-dependent.
>
> **destroy[s]** destroys **s** in an orderly way, freeing the space it occupies. Note that this has nothing to do with deleting any underlying data structures or processes associated with the stream (like a disk file, for example, or the keyboard process).

Each of these operations is defined more precisely in the descriptions of the individual stream types which appear separately. All of the stream routines produce the **StreamType** error when the variant of the **StreamObject** they are passed is not what they are expecting. See the **Disk Streams**, **Display**, and **Keyboard** sections for details of specific stream types. For each **StreamType**, there is a create procedure which returns a **StreamHandle**. To invoke **get** for a **StreamHandle**, **s**, simply say **s.get[s]**.

The **Other** variant of a **StreamObject** is provided so that clients can easily provide other types of streams using the same standard set of operations. The **data** field of an **Other StreamObject** should point to any additional data required by the particular stream. Clients with more than one type of **Other** stream should include a type code in the type field.

Alto/Mesa String Package

April 1979

This module contains procedures that implement various string operations. The necessary TYPE and PROCEDURE declarations appear in **StringDefs** and are described below. (Constants defining word size, character size, etc. are in **AltoDefs**.) Specific language features concerning STRINGs are described in more detail in the *Mesa Language Manual* (see the index).

```
SubStringDescriptor: TYPE = RECORD [
    base: STRING,
    offset, length: CARDINAL];

SubString: POINTER TO SubStringDescriptor;
```

A **SubStringDescriptor** describes a region within a string. The first character is **base[offset]** and the last character is **base[offset+length-1]**.

```
WordsForString: PROCEDURE [nchars: CARDINAL] RETURNS [CARDINAL];
```

Calculates the number of words of storage needed to hold a string of length **nchars**. The value returned includes any system overhead for string storage.

```
LowerCase: PROCEDURE [CHARACTER] RETURNS [CHARACTER];
```

Changes upper case characters into lower case ones. This is a noop if the character is not a letter.

```
UpperCase: PROCEDURE [CHARACTER] RETURNS [CHARACTER];
```

Changes lower case characters into upper case ones. This is a noop if the character is not a letter.

String Construction

```
AppendChar: PROCEDURE [s: STRING, c: CHARACTER];
```

Appends the character **c** to the end of the string **s**; **s.length** is updated; **s.maxlength** is unchanged.

```
AppendString: PROCEDURE [to, from: STRING];
```

Appends the string **from** to the end of the string **to**; **to.length** is updated; **to.maxlength** is unchanged.

AppendSubString: PROCEDURE **[to:** STRING, **from: SubString];**

Appends the substring in **from** to the end of the string in **to**; **to.length** is updated: **to.maxlength** is unchanged.

StringBoundsFault: SIGNAL **[s:** STRING] RETURNS **[ns:** STRING];

An attempt was made to increase the length of **s** to be larger than **s.maxlength**. The catch phrase should return a string **ns** with more room.

DeleteSubString: PROCEDURE **[s: SubString];**

Deletes the substring described by **s** from the string **s.base**; **s.base.length** is updated; **s.base.maxlength** is unchanged.

String Comparison

EqualString, EqualStrings: PROCEDURE **[s1, s2:** STRING] RETURNS **[**BOOLEAN**];**

Returns TRUE if **s1** and **s2** contain exactly the same characters.

EquivalentString, EquivalentStrings: PROCEDURE **[s1, s2:** STRING] RETURNS **[**BOOLEAN**];**

Returns TRUE if **s1** and **s2** contain the same characters except for case shifts. *Note: strings containing control characters may not be compared correctly.*

EqualSubString, EqualSubStrings: PROCEDURE **[s1, s2: SubString]** RETURNS **[**BOOLEAN**];**

Analogous to **EqualString** and **EqualStrings**.

EquivalentSubString, EquivalentSubStrings: PROCEDURE **[s1, s2: SubString]** RETURNS **[**BOOLEAN**];**

Analogous to **EquivalentString** and **EquivalentStrings**.

String to Binary Conversion

StringToNumber: PROCEDURE **[s:** STRING, **radix:** CARDINAL] RETURNS **[**UNSPECIFIED**];**

The characters of **s** are interpreted as a number whose value is returned. **radix** is used in the conversion unless the "**B**" or "**D**" notation is used to force octal or decimal. Supplying **radix** values of other than 8 or 10 is not supported.

StringToDecimal: PROCEDURE **[s:** STRING] RETURNS **[**INTEGER**];**

Calls **StringToNumber[s, 10]**.

StringToOctal: PROCEDURE [s: STRING] RETURNS [UNSPECIFIED];

Calls **StringToNumber[s, 8]**.

StringToLongNumber: PROCEDURE [s: STRING, **radix**: CARDINAL] RETURNS [LONG INTEGER];

The characters of **s** are interpreted as a LONG INTEGER whose value is returned. **radix** is used in the conversion unless the "**B**" or "**D**" notation is used to force octal or decimal. Supplying **radix** values of other than 8 or 10 is not supported.

InvalidNumber: SIGNAL;

A string is not a valid number if it is empty or contains characters other than digits (a leading '- and trailing '**B** or '**D** with scale factor are allowed).

Binary to String Conversion

AppendNumber: PROCEDURE [s: STRING, **n**, **radix**: CARDINAL];

The value of **n** is converted to text using **radix** and appended to **s**; **radix** should be in the interval **[2..36]**.

AppendDecimal: PROCEDURE [s: STRING, **n**: INTEGER];

IF **n** < 0 THEN **AppendChar[s, '-]; AppendNumber[s,** ABS**[n], 10]**.

AppendOctal: PROCEDURE [s: STRING, **n**: UNSPECIFIED];

AppendNumber[s, n, 8]; AppendChar[s, 'B].

AppendLongDecimal: PROCEDURE [s: STRING, **n**: LONG INTEGER];

Appends '- to **s** if **n** is negative. It then calls **AppendLongNumber** with a **radix** of 10.

AppendLongNumber: PROCEDURE [s: STRING, **n**: LONG UNSPECIFIED, **radix**: CARDINAL];

The value of **n** is converted to text using **radix** and appended to **s**; **radix** should be in the interval **[2..36]**.

Alto/Mesa Time Package

April 1979

TimeConvert contains procedures that implement various operations on dates and times. The necessary TYPE and PROCEDURE declarations appear in **TimeDefs** and are described below.

PackedTime: TYPE = LONG CARDINAL;

A **PackedTime** is the number of seconds since midnight January 1, 1901 GMT.

DefaultTime: PackedTime;

```
UnpackedTime: TYPE = RECORD [
    year: [0..2050],  -- years less than 1901 are not possible
    month: [0..12),   -- January = 0
    day: [0..31],     -- first day of month = 1
    hour: [0..24),
    minute: [0..60),
    second: [0..60),
    weekday: [0..6],  -- Monday = 0
    zone: [-12..12],  -- Pacific = 8
    dst: BOOLEAN];
```

CurrentDayTime: PROCEDURE RETURNS **[time: PackedTime];**

Returns the current date and time in packed format.

UnpackDT: PROCEDURE **[PackedTime]** RETURNS **[time: UnpackedTime];**

Converts a packed format time to a more convenient unpacked format. If the argument is **DefaultTime**, the current time is used.

PackDT: PROCEDURE **[unp: UnpackedTime, computeDST:** BOOLEAN**]**
RETURNS **[time: PackedTime];**

Converts **unp** into packed format. If **computeDST** is TRUE, the time zone and daylight savings time are computed according to local conventions rather than taken from **unp**. If any of the fields of **unp** contain illegal values, **PackDT** will signal **InvalidTime.**

InvalidTime: ERROR;

PackDT has discovered illegal values in the unpacked time.

AppendDayTime: PROCEDURE **[s:** STRING**, u: UnpackedTime];**

Converts **u** to a text string of the form 12-Jan-78 14:56:13 and appends it to **s.**

AppendFullDayTime: PROCEDURE **[s:** STRING, **u: UnpackedTime];**

Like **AppendDayTime** except that the time zone is included, *e.g.* 1-May-78 14:56:13 PDT.

Alto/Mesa Traps

April 1979

All traps generated by Mesa are converted into signals or errors of the same name. Except for the parity error and stack error traps, all of the signals described below are related to specific language features and are described in more detail in the *Mesa Language Manual* (see the index).

StartFault: SIGNAL [dest: GlobalFrameHandle];

An attempt was made to start or restart the frame **dest**, but it is not a valid global frame. Usually this means that a module or program was not bound, a program being restarted does not STOP, or a module that has parameters is being started as a result of a start trap.

ControlFault: SIGNAL [source: FrameHandle] RETURNS [ControlLink];

An attempt was made to transfer to a null control link while executing in the frame **source**. Usually this means some external links have been clobbered. This signal can be resumed with a control link that will be used to retry the transfer.

UnboundProcedure: SIGNAL [dest: ControlLink] RETURNS [ControlLink];

An attempt was made to transfer to **dest** but it had an unbound tag. Usually this means some external links have not been bound or the GFT entry of **dest** is null (probably a deleted module). This signal can be resumed with a control link that will be used to retry the transfer.

BoundsFault: SIGNAL;

An attempt has been made to exceed the bounds of a STRING, ARRAY or **SubRange** (note that this includes assigning a negative INTEGER to a CARDINAL.) in a module that was compiled with the /b switch.

PointerFault: SIGNAL;

An attempt has been made to dereference NIL in a module that was compiled with the /n switch.

LinkageFault: ERROR;

A transfer has been attempted through a port that has not been connected to some other port or procedure (the link field of the port was null).

PortFault: ERROR;

A transfer has been attempted to a port which is not pending (the frame field of the destination port is null). This error is used to handle the startup transients common in a configuration of coroutines.

ParityError: SIGNAL **[address:** POINTER**];**

A parity error has occured in the word pointed to by **address.**

PhantomParityError: SIGNAL**;**

The parity error process was started, but a sweep through memory found no errors.

StackError: SIGNAL **[FrameHandle];**

The stack has either overflowed or underflowed while executing in the designated frame. Usually this means that either some code has been smashed or some external link is incorrect.

Fine point: external links may be incorrect because the user forgot to rebind after a recompilation, or procedure variables were incorrectly LOOPHOLEd.