For Xerox Internal Use Only -- December 15, 1980

ALTO OPERATING SYSTEM

REFERENCE MANUAL

Compiled on: December 15, 1980

Xerox Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, California 94304

Alto Operating System Reference Manual

OS version 19/16

1. Introduction

This manual describes the operating system for the Alto. The manual will be revised as the system changes. Parts of the system which are likely to be changed are so indicated; users should try to isolate their use of these facilities in routines which can easily be modified, or better yet, avoid them entirely, if possible.

The system and its description can be separated into two parts:

a) User-callable procedures, which are of two kinds: standard procedures which are always provided, and library procedures which must be loaded with the user's program if they are desired. This manual describes only standard procedures; the library procedures are documented in the "Alto Packages Manual."

b) Data structures, such as disk files and directories, which are used by the system but which are also accessible to user procedures and subsystems.

The system is written almost entirely in Bcpl. Its procedures are invoked with the standard Bcpl calling sequence, and it expects the subsystems it calls to be in the format produced by the Alto Bcpl loader.

2. Hardware summary

This section provides an overview of the Alto Hardware. Briefly, every Alto has:

a) A memory of 64k words of 16 bits each. The cycle time is 850ns.

b) An emulator for a standard instruction set.

c) Secondary memory, which may consist of one or two Diablo 31 cartridge disk drives, or one Diablo 44 cartridge disk drive. The properties of these disks are summarized in Table 2.2.

d) An 875 line TV monitor on which a raster of square dots can be displayed, 606 dots wide and 808 dots high. The display is refreshed from Alto memory under control of a list of display control blocks. Each block describes what to display on a horizontal band of the screen by specifying:
- the height of the band, which must be even;
- the width, which must be a multiple of 32; the space remaining on the right is filled with background;
- The indentation, which must be a multiple of 16; the space thus reserved on the left is filled with background;
- the color of the background, black or white;
- the address of the data (must be even), in which 0 bits specify background. Each bit controls the color of one dot. The ordering is increasing word addresses and then bit numbers in memory, top to bottom and then left to right on the screen; and a half-resolution flag which makes each dot twice as wide and twice as high.

There is also a 16 x 16 cursor which can be positioned anywhere on the screen. If the entire screen is filled at full resolution, the display takes about 60% of the machine cycles and 30704D words of memory.

e) A 44-key keyboard, 5-finger keyset, and mouse

f) A Diablo printer interface

g) An Ethernet interface

h) Interfaces for analog-to-digital and digital-to-analog conversion, for TV camera input, and for a RS-232b (teletype) connection

i) A real-time clock and an interval timer (see table 2.1 for brief descriptions)

3. User-callable procedures

This section describes the operating system facilities provided by procedures which can be called from user programs using the standard Bcpl calling sequence. All of these procedures are a permanent part of the operating system, automatically available to any user program.

Although this manual describes a rather extensive set of facilities, which together occupy close to 12K words of memory, portions of the system can be deactivated (see Junta), thus freeing the memory they use. When the user program finishes execution, the deactivated portions can be retrieved from the disk and reinitialized.

Default arguments: Many of the procedures given below have rather long argument lists, but have convenient defaulting schemes. The documentation decorates argument lists with default values. An argument followed by [exp] will default if omitted or zero to the value exp; an argument followed by [...exp] will default if omitted to exp. Although Bcpl allows you to omit procedure arguments by using "nil," the called procedure cannot detect its use; it therefore cannot be the basis for defaulting arguments.

3.1. Facilities

The facilities of the operating system fall into fairly neat categories; often this is because the operating system has simply loaded a standard library subroutine as part of its environment. This manual offers summarized documentation for the functions in the various software "packages;" more documentation can be found in the "Alto Software Packages Manual." (Note: Appendices to this manual include documentation of the packages most relevant to the operating system.) In outline, the operating system provides:

- A "basic" resident that maintains a time-of-day clock, that processes parity error interrupts, and that contains the resident required to interface to Swat, the debugger.

- The Bcpl runtime support module, which provides several functions (such as a stack frame allocator) that are necessary to permit Bcpl programs to run.

- Disk drivers for transferring complete pages between memory and existing files on the disk. This is the BfsBase package.

- Disk drivers for creating new files, and for extending or shortening existing files. This is the BfsWrite package.

- A simple storage allocator for managing "zones" of working storage. This is the Alloc package.

- Disk "streams," which implement sequential byte or word I/O to the disk. This is the DiskStreams package.

- Disk directory management, which provides facilities for searching directory files for entries that associate a string name and a disk file.

- A keyboard handler, which decodes keyboard interactions into a sequence of ASCII characters.

- A display driver, which maintains a "system display," and handles the printing of characters on the display. This is the DspStream package.

- Miscellaneous functions, including (1) the "call subsystem" function, which reads a file produced by the Bcpl loader into memory and executes it; (2) allocation functions that manage the space not used by the operating system or the user code, providing a stack for the user program and fixed-size blocks that it may require; (3) the procedure for de-activating various portions of the operating system; and (4) additional utilities.

3.2. Loading and Initialization

The facilities of the operating system are made accessible to user programs via static variables that refer to system procedures or system scalars. Because these objects are not defined in your Bcpl program, you must declare the names to be external. The Bcpl loader, Bldr, automatically reads the file Sys.Bk, which describes how to arrange that your program's external references will match up with the operating system objects (for details, see Bldr documentation in the Bcpl manual). This arrangement does not require re-loading programs when objects in the operating system move.

When a Bcpl program is read into the Alto memory, all of the system procedures described below will have been initialized. A region is reserved for allocating system objects (e.g., disk streams); currently, about 6 disk streams or equivalent can be accomodated. If the space reserved is inadequate for your application, the system zone can be replaced with one constructed by your program. In addition, most procedures that create system objects have provision for an optional "zone" argument used for seizing space (see section 4.5).

3.3. Errors

Whenever the system detects an error for which the user program has not supplied its own error routine, the call SysErr(p1, errCode, p2, p3, ...) is executed. The errCode is a number that identifies the error; the p's are parameters that add details.

Normally, SysErr calls Swat (the debugger), which will print out an intelligible error message retrieved from the file Sys.Errors. The facilities of Swat (see "Alto Subsystems Manual") can then be used to interrogate the program state more fully, and ultimately to continue or abort its execution.

3.4. Streams

The purpose of streams is to provide a standard interface between programs and their sources of sequential input and sinks for sequential output. A set of standard operations, defined for all streams, is sufficient for all ordinary input-output requirements. In addition, some streams may have special operations defined for them. Programs which use any non-standard operations thereby forfeit complete compatibility.

Streams transmit information in atomic units called items. Usually an item is a byte or a word, and this is the case for all the streams supplied by the operating system. Of course, a stream supplied to a program must have the same ideas about the kind of items it handles as the program does, or confusion will result. Normally, streams which transmit text use byte items, and those which transmit binary information use words. (The 16-bit quantity which Bcpl passes as an argument or receives as a result of a stream operation could be a pointer to some larger object such as a string, although the operating system implements no such streams. In this case, storage allocation conventions for the objects thus transmitted would have to be defined.)

You are free to construct your own streams by setting up a suitable data structure (section 4.2) which provides links to your own procedures which implement the standard operations.

The standard operations on streams are (S is the stream; "error" means that Errors(S, ec) is executed, where ec is an error code):

Gets(S)	returns the next item. Some streams give an error if Endofs(S) is true before the call, and others just wait for the next item.
Puts(S, I)	writes I into the stream as the next item; error if the stream is read-only, if there is no more space or if there is some hardware problem.
Resets(S)	restores the stream to some initial state, generally as close as possible to the state it is in just after it is created.
Putbacks(S, I)	modifies S so that the next Gets(S) will return I and leave S in the state it was in before the Putbacks. Error if there is already a putback in force on S. (No streams provided by the operating system implement a Putbacks operation.)
Endofs(S)	true if there are no more items to be gotten from S. Not defined for output streams.
Closes(S)	destroys S in an orderly way, and frees the space allocated for it. Note that this has nothing to do with deleting a disk file.
Stateofs(S)	returns a word of state information which is dependent on the type of stream.
Errors(S, ec)	reports the occurrence of an error with error code ec on the stream. When a system stream is created, Errors is initialized to SysErr (see section 3.3), but the user can replace it with his own error routine.

Streams are created differently depending on the device being accessed (disk, display, keyboard, or memory). The procedures for creating streams are described below.

3.4.1. Disk streams

The system distinguishes four kinds of object which have something to do with storing data on the disk:

Disk Pack:	A storage medium that is capable of storing data in various pages. Most operating system functions default the choice of disk to "sysDisk", a structure which describes drive 0 of a Diablo model 31 cartridge.
Disk file:	A vector of bytes of data held on some disk, organized into pages for some purposes. A file exists only on the disk (except that parts of it may be in memory if an output stream is associated with it) and is named by an 80-bit entity called a file pointer (FP).
File directory:	A disk file which contains a list of pairs <string name, FP>. Documentation on the format of the file can be found with the BFS package documentation contained in an appendix to this manual.
Disk stream:	Used by a program to transfer information to or from a disk file. A stream exists only in memory and is named by a pointer to a data structure.

The procedures that operate on disk streams are described in documentation for the "DiskStreams" software package contained in an appendix to this manual. Below is a summary list of the functions (in addition to the generic functions described above):

CreateDiskStream(filePtr, type [ksTypeReadWrite], itemSize [wordItem], Cleanup [Noop], errRtn [SysErr], zone [sysZone], nil, disk [sysDisk]) = a disk stream, or 0 if an error is encountered while initializing the stream. filePtr is the sort of object stored in a file directory. Legal types are ksTypeReadOnly, ksTypeReadWrite, and ksTypeWriteOnly. Legal item sizes are wordItem and charItem.

CleanupDiskStream(s) Flush any buffers to the disk.

ReadBlock(s, address, count) = actualCount. Read up to count words from the stream into consecutive memory locations; return the actual number of words read. (Non-intuitive things happen at the end of a file with an odd number of bytes -- read the documentation carefully)

WriteBlock(s, address, count) Write count words from consecutive memory locations onto the stream.

LnPageSize(s) = log (base 2) of the page size, in words, of the files manipulated by the stream.

PositionPage(s, page) Positions the file to byte 0 of the specified page (page 1 is the first data page).

PositionPtr(s, byteNo) Positions the file to the specified byte of the current page.

FileLength(s, filePos []) = Length. Returns number of bytes in file; positions stream to the last byte.

FilePos(s, filePos []) = Pos. Returns the current byte position in the file.

SetFilePos(s, filePos) or SetFilePos(s, HighOrder, LowOrder) Sets the position of the file to the specified byte.

GetCurrentFa(s, fileAddress) Returns the current file address.

JumpToFa(s, fileAddress) Positions the file to the specified address (usually obtained from GetCurrentFa).

GetCompleteFa(s, completeFileAddress) Returns a complete file address, including a filePtr.

TruncateDiskStream(s) Truncates the file to the current position.

ReadLeaderPage(s, address) Reads the 256-word leader page of the file into consecutive locations starting at address.

WriteLeaderPage(s, address) Writes 256 words onto the leader page of the file.

The operating system also contains a package for dealing with files at a lower level, the "Bfs" (Basic file system) package.

Disk Errors: The system will repeat five times any disk operation which causes an error. On the last three repetitions, it will do a restore operation on the disk first. If five repetitions do not result in an error-free operation, a (hard) disk error occurs; it is reported by a call on Errors for the stream involved.

3.4.2. Display streams

Display streams are implemented with the "DspStream" package, described in separate documentation contained in an appendix to this manual. Below is a list of the functions included (in addition to the generic stream functions):

CreateDisplayStream(nLines, pBlock, lBlock, Font [sysFont], wWidth [38], options [DScompactleft+DScompactright], zone [sysZone])	= a display stream. pBlock is the address of a region lBlock words long for the display bitmap. nLines is the number of text lines in the stream. This procedure does not commence displaying the stream text -- see ShowDisplayStream.
ShowDisplayStream(s, how [DSbelow], otherStream [dsp])	This procedure controls the presentation of the stream on the screen. If how is DSbelow, the stream will be displayed immediately below otherStream; if DSabove, immediately above; if DSalone, the stream will become the only display stream displayed. If how is DSdelete, the stream s will be removed from the display. For DSalone and DSdelete, the third argument is needless.
GetFont(s)	Returns current font.
SetFont(s, font)	Sets current font (use carefully -- see documentation).
ResetLine(s)	Erases all information on the current line and resets the position to the left margin.
GetBitPos(s)	Returns the horizontal position of the stream.
SetBitPos(s, pos)	Sets the horizontal position on the current line (use carefully -- see documentation).
GetLinePos(s)	Returns the index of the line into which characters are presently being put.
SetLinePos(s, pos)	Sets the line number into which subsequent characters will be put.
InvertLine(s, pos)	Inverts the black/white sense of the line given by pos.
EraseBits(s, nBits, flag [0])	Erase bits moving forward (nBits>0) or backward (nBits<0) from the current position. Set to background if flag=0; to the complement of the background if flag=1; invert present values if flag=-1.
GetLmarg(s); SetLmarg(s)	Get and set left margin for the current line.
GetRmarg(s); SetRmarg(s)	Get and set right margin for the current line.
CharWidth(StreamOrFont, char)	Get the width of the character, using the specified font or the current font in the specified stream.

The "system display stream" is always open, and can be accessed by the system scalar "dsp."

3.4.3. Keyboard Streams

There is a single keyboard stream in which characters are buffered. The stream is always open, and may be accessed through the system scalar "keys." The only non-null operations are Gets; Endofs, which is true if no characters are waiting; and Resets, which clears the input buffer.

The keyboard handler periodically copies the mouse coordinates into the cursor coordinates, truncating at the screen boundary. This function is governed by the value of a cell referenced by @ lvCursorLink; if it is zero, the function is disabled.

Low-level keyboard functions. Although the standard keyboard handler contains no facilities for detecting transitions of keyset or mouse keys, a user function may be provided that will be called 60 times a second and can extract relevant information from a table passed to it. The call SetKeyboardProc(uKbProc, stack, stackLength) will install uKbProc as the user procedure; stack is a vector that will be used for stack space when uKbProc is run (you must provide enough!). SetKeyboardProc() will reset the keyboard handler, and cease calling uKbProc. (Note: If the program has used the Junta procedure, the user keyboard procedure must be deactivated during a CounterJunta or finish unless all its state lies below OsFinishSafeAdr.) If active, every 16 milliseconds, the keyboard handler will execute uKbProc(tab), where tab points to a data structure defined by the KBTRANS structure (see the file SysDefs.d). The Transition word is non-zero if a key transition has been detected; GoingUp or GoingDown tell which sort of transition has occurred; and KeyIndex gives the key number. KeyState is a 5-word table giving the state of the keys after the transition has occurred: if a key with KeyIndex=i is presently down, bit (i rem 16) of word (i div 16) will be 1. The entries CursorX and CursorY give the current location of the cursor.

The value returned by uKbProc determines subsequent processing. If true is returned, the operating system treats the key transition (if any) according to normal conventions. If false is returned, the operating system assumes that uKbProc has performed whatever processing is intended, and the interrupt is simply dismissed.

KeyIndex values are tabulated below. Keys are normally given by their lower-case marking on the key top; those with more than one character on their tops are specified by <name>. <X> are unused bits; <blank-top> is the key to the right of the <bs> key; <blank-middle> to the right of <return>; and <blank-bottom> to the right of <shift-right>.

Values Keys

0-15 5 4 6 e 7 d u v 0 k - p / \ <lf> <bs>
16-31 3 2 w q s a 9 i x o l , ’] <blank-middle> <blank-top>
32-45 1 <esc> <tab> f <ctrl> c j b z <shift-left> . ; <return> _ <del> <X>
48-63 r t g y h 8 n m <lock> <space> [= <shift-right> <blank-bottom> <X> <X>
64-71 unused
72-76 Keyset keys in order, left=72; right=76
77 RED (or left or top) mouse button
78 BLUE (or right or bottom) mouse button
79 YELLOW (or middle) mouse button

As an aid to interpreting KeyIndex values, the system scalar kbTransitionTable points to a table, indexed by KeyIndex, that gives a KBKEY structure for the key; if it is zero, the operating system has no standard interpretation of the key.

3.4.4. Fast Streams to Memory

The operating system also contains procedures that allow very efficient stream I/O to memory blocks. These functions, described in the Streams package documentation, allow one for example to use much more memory buffering for disk transfers than normally allocated by the disk stream mechanism.

3.5. Directory Access

Most user programs do not concern themselves with file pointers, but use system routines which go directly from string names to streams. By a "file name" we mean a string which can be converted into a file identifier by looking it up in a directory. File names are arbitrary Bcpl strings which contain only upper and lower case letters, digits, and characters in the string "+-.!$". File names are stored in directories as they are typed, but no distinction is made between upper and lower case letters when they are looked up. Dots (".") are used to separate file names into parts. If there is more than one part, the last part is called the extension, and is conventionally used much like extensions in Tenex.

There is an optional version number facility. It is not available in the standard release of the operating system (NewOs.boot), but is available in an unsupported alternate version (NewOsV.boot). If the version number facility is enabled, the interpretation of exclamation mark ("!") is special; if a file name ends with a ! followed only by digits, the digits specify the file version number.

A lookup name, presented to one of the directory functions given below, is usually a file name. However, it may optionally specify the name of a directory in which to look for the file (or record the new file). The lookup name is processed from left to right. If the character "<" appears at the head of the lookup name, the system directory ("SysDir.") becomes the "current" directory; whenever the character ">" follows a name, the name is looked up in the current directory and that file becomes the new current directory. If no directory is specified in the lookup name, the "working directory" is assumed. Example: "<dir>fil." will look up dir in the system directory SysDir, and will then look up fil in dir. Any illegal characters in a lookup name are replaced with "-" characters.

File Versions: The file system also supports multiple versions of the same file; this feature may be enabled or disabled when the operating system is installed. The version number is recorded by appending an exclamation mark and the decimal version number to the file name; file names without version numbers appended act as if they are "version 0." The OpenFile function uses lookup names and version control information to locate a desired file. If the lookup name contains a version number (e.g., "Sys.Errors!3."), then no version defaulting is done--the lookup operates on precisely the file specified. (This processing is identical with versions enabled and disabled.)

If the lookup name does not specify a version number and file versions are enabled, then the versionControl parameter specifies how defaulting is to be done (in the definitions, "oldest" refers to the file with the "lowest" version number; "latest" refers to the file with the "highest" version number):

verLatest — The latest version is used.

verLatestCreate — The latest version is used. If the file does not exist, it is created with version number 0 (i.e., no number will be appended explicitly to the file name): this is to prevent needless accumulation of version numbers in system-related files (.e.g, .Run files).

verOldest — The oldest version is used.

verNew — A new file will always be created. A system parameter, established when the system is installed, determines how many old versions will be preserved. If that default should be overriden, just add the desired number of versions to verNew, e.g. a versionControl value of verNew+4 will create a new file and retain at most three older versions.

This version option may reuse disk pages allocated for the oldest version of the file, but the serial number and file name will of course be changed. If (newest-oldest)+1 is greater than or equal to the number of versions to keep, oldest is reused in this fashion to become version newest+1. For example, if verNew is specified, 2 versions are to be kept, and foo!2 and foo!3 exist, verNew will create the file foo!4 by remaking the old file foo!2. Note that this calculation does not verify that all versions between oldest and newest actually exist.

If only one file matches the lookup name, and its version number is 0, the file is simply overwritten (like verLatestCreate); a new version is not created.

If no files of the given name exist, version number 0 of the file is created (i.e., no version number is explicitly attached to the file name). The verNewAlways option (below) can be used if version 1 should be created.

verNewAlways Similar to verNew, but if no earlier version of the file exists, version 1 is created.

If versions are not enabled, then exact matches are performed on the entire file name. Thus, if the file "Sys.Errors!2" is present on a disk with versions disabled, the lookup name "Sys.Errors" will not match this file; the lookup name "Sys.Errors!2" will. The versionControl parameter is still relevant: if no file matching the lookup name is found, verLatest and verOldest will not create a new file, whereas the other versionControls will.

The following function creates a disk stream (see above) in conjunction with the Alto directory structure:

OpenFile(lookupname, ksType [ksTypeReadWrite], itemSize [wordItem], versionControl [if ksType=ksTypeReadOnly then verLatest else if ksType=ksTypeWriteOnly then verNew else verLatestCreate], hintFp [0], errRtn [SysErr], zone [sysZone], nil, disk [sysDisk], CreateStream [CreateDiskStream]) = a disk stream, open on the specified file, or 0 if the open is unsuccessful for some reason. This routine parses the lookup name, searching directories as needed. After applying version control (e.g., making a new version), it calls CreateStream(filePointer, ksType, itemSize, Noop, errRtn, zone, nil, disk), and returns the value of that call.

If hintFp is provided, it is assumed to be a file pointer (FP) that "hints" at the correct identification of the file. Before searching a directory, OpenFile will try using the hint to open the file, quickly returning a stream if the hint is valid (though no name or version checking is done). If the hint fails and lookupname is non-zero, the name will be parsed and looked up in the normal fashion. hintFp will be filled in with the correct file pointer. Note: If you wish to use standard file-lookup procedures, but to have the FP for the resulting file returned to you, zero the hintFp vector before calling OpenFile. In this case, the value of hintFp is not used in the lookup, but is filled in with the results.

OpenFileFromFp(hintFp) = OpenFile(0, 0, 0, 0, hintFp)

DeleteFile(lookupname, versionControl [verOldest], errRtn [SysErr], zone [sysZone], nil, disk [sysDisk]) = success. Deletes the file on the disk and removes the corresponding entry from the directory specified in lookupname. Returns "true" if a file was correctly found and deleted, otherwise "false."

SetWorkingDir(name, fp, disk [sysDisk]) Sets the "current" directory for further lookups on the given disk. When the system is booted, the current directory is set to "<SysDir."

3.5.1. Lower-level directory functions

Several functions are provided for those who wish to deal with directories and file names at a lower level. The format of an Alto file directory is documented in the Disks documentation; definitions appear in AltoFileSys.d.

ParseFileName(destName, srcName, list, versionControl) = stream or 0. Strips leading directory information from srcName, puts the result in destName, appending a "." if necessary, and returns a stream open on the directory in which the file should be looked up. list!0 = an errorRoutine, list!1 = a zone, list!3 = a disk which will be passed to OpenFile along with versionControl when opening the directory stream.

FindFdEntry(s, name, compareFn [0], dv [], hd [], versionControl [verLatest], extraSpace [0]) = a word pointer into the stream s of a directory entry, or -1 if no entry is located. If compareFn is 0, normal comparison of file names and version control is performed; the result is a directory entry in dv, and a hole descriptor (hd) for a hole large enough to include the name, a new version number, and extraSpace words.

Otherwise, compareFn is a user procedure that is invoked as each file name is read from the directory: compareFn(name, nameRead, dvRead). nameRead is the Bcpl name extracted from the directory; dvRead is the dv extracted from the directory; and name is simply the second argument passed to FindFdEntry (which need not be a string). If compareFn returns false, the directory scan halts; the value of FindFdEntry is the byte position in the stream. If compareFn returns true, the search proceeds.

Strategic note: If compareFn is TruePredicate, the directory is simply scanned in order to locate a hole large enough for extraSpace words. The result is saved in the hd hole descriptor, which may be passed to MakeNewFdEntry.

In the standard release of the operating system (version numbering absent), the directory stream is left positioned at the matching directory entry if one was found and at the position described by hd otherwise.

MakeNewFdEntry(s, name, dv, hd, extraStuff) makes a directory entry: dv is a pointer to a DV structure for the first part of the entry; name is a Bcpl string that is recorded after the entry (this string must be a legal internal file name, with the dot "." appended), and extraStuff is a pointer to a vector of additional stuff that will be entered following the name. The hd parameter is a pointer to a "hole descriptor" as returned from FindFdEntry.

DeleteFdEntry(s, pos) Deletes the directory entry at byte location pos of the directory open on stream s.

StripVersion(string) = version number. This function strips a version number, if any, from the end of the string argument, and returns the number (0 if no version specified). If, after stripping, there is no final "." on the string, one is appended.

AppendVersion(string, version) Appends a version number and final "." to the string.

WriteDiskDescriptor()	If changes have occurred, the copy of the disk descriptor for sysDisk that resides in memory is written onto the disk file "DiskDescriptor."
ReadDiskDescriptor()	This function restores the copy of the disk descriptor for sysDisk that resides in memory from the disk file "DiskDescriptor."

3.6. Memory management

Table 3.1 shows the layout of memory. Table 3.2 tells how to obtain the current values of the symbolic locations in Table 3.1. The free space (EndCode to StackEnd) can be manipulated as follows:

GetFixed(nwords)	returns a pointer to a block of nwords words, or 0 if there isn't enough room. It won't leave less than 100 words for the stack to expand.
FreeFixed(pointer)	frees a block provided by GetFixed.
FixedLeft()	returns the size of the biggest block which GetFixed would be willing to return.
SetEndCode(newValue)	resets endCode explicitly. It is better to do this only when endCode is being decreased.

The allocator is not very bright. FreeFixed decrements endCode if the block being returned is immediately below the current endCode (it knows because GetFixed puts the length of the block in the word preceding the first word of the block it returns; please do not rely on this, however, since there is no guarantee that later allocators will use the same scheme). Otherwise it puts the block on a free list. When another FreeFixed is done, any blocks on the free list which are now just below endCode will also be freed. However, the allocator makes no attempt to allocate blocks from the free list.

3.7. The Alloc allocator

The operating system includes a copy of the Alloc package; documentation is contained in an appendix to this manual.

InitializeZone(start, length, OutOfSpaceRoutine [...SysErr], MalFormedRoutine [...SysErr]) = a "zone." These zones are compatible with the "zone" arguments to operating system functions (e.g., sysZone). Allowing MalFormedRoutine to default to SysErr causes a through check of the zone data structures to be performed each time a block is allocated or freed. To avoid this (considerable) overhead, pass a zero for the MalFormedRoutine. The default sysZone has a MalformedRoutine of SysErr.

AddToZone(zone, block, length) Adds block to the zone.

Allocate(zone, length, returnOnNoSpace [false], even [false]) = pointer to a block of length words allocated from zone. If even is true, the pointer is guaranteed to be a even number.

Free(zone, ptr)	Returns the block pointed to by ptr to the zone.
CheckZone(zone)	Performs a consistency check on the zone data structure.

3.8. The Basic File System

A set of procedures for driving the disk hardware for Diablo Model 31 and 44 disk cartridges is included in the operating system. These functions are documented in the "Disks" documentation, appended to this manual.

3.9. Objects

It is often convenient to define an abstract object and its operations by a single entity in the Bcpl language. As the largest entity Bcpl can deal with is a 16-bit number, we must use a pointer to a structure of some kind that defines both the procedures and data associated with the object. Streams, Zones and Disks are examples of such abstract objects. Such objects are typically defined by a structure such as:

```
structure ZN:
    [
    Allocate    word   //Op
    Free        word   //Op
    Base        word   //Val
    Length      word   //Val
    ]
```

where the Op's point to procedures and the Val's are data for the structure. A typical call on one of the abstract procedures is thus (zone>>ZN.Allocate)(zone, arg1, arg2, arg3). The virtue of such an arrangement is that any structure that simulates the effects of the procedures can pose as a Zone.

In order to encourage the use of such objects, the operating system has very efficient implementations for this calling mechanism:

Call0(s, a, b, ...)	Does (s!0)(s, a, b, ...)
Call1(s, a, b, ...)	Does (s!1)(s, a, b, ...)

Call2, Call3, ..., Call15 analogously.

Thus, the operating system defines Allocate=Call0, and Free=Call1, consistent with the Alloc package described above. Note for assembly-language programmers: the CallX functions actually enter the proper function at the <u>second</u> instruction, having already executed a STA 3 1,2 to save the return address.

3.10. Miscellaneous

This section describes a collection of miscellaneous useful routines:

Wss(S, string)	writes the string on stream S.
Ws(string)	writes the string on the system display stream, dsp.
Wl(string)	Ws(string), followed by a carriage return.
Wns(S, n, nc [0], r[-10])	writes a number n to stream S, converting using radix abs(r). At least nc characters are delivered to the stream, using leading spaces if necessary. The number is printed in signed notation if r<0, in unsigned notation if r>0.
Wos(S, n)	writes an unsigned octal representation of n on stream S.
Wo(n)	writes an unsigned octal representation of n on the display stream.

TruePredicate() always returns -1.

FalsePredicate() always returns 0.

Noop() null operation; returns its first argument if any.

Dvec(caller, nV1, nV2, ...) this routine allocates "dynamic" vectors in the current frame. caller is the name of the procedure calling Dvec. The use of the routine is best given with an example: the routine ShowOff wants two vectors, V1 and V2:

```
let ShowOff(V1length, V2length) be
  [
  let V1 = V1length
  let V2 = V2length
  Dvec(ShowOff, lv V1, lv V2)
  // now V1 points to a block V1length+1 words long
  // and V2 points to a block V2length+1 words long
  ]
```

Warning: any addresses that point into the stack frame of ShowOff before it is moved by the Dvec call will not be correct after the call. Thus, for example, a "let a = vec 10" before the call will cause the address in a to be useless after the call.

DefaultArgs(lvNa, base, dv1, dv2,.....) Utility procedure to fill in default arguments. lvNa points to the "numargs" variable in the procedure; abs(base) is the number of initial arguments that are not to be defaulted; the dv_i are the default values (i<11). If base<0, then an actual parameter of zero will cause the default to be installed; otherwise only (trailing) omitted parameters are defaulted. Thus:

```
let Mine(how, siz, zone, errRtn; numargs n) be
  [
  DefaultArgs(lv n, -1, 100, sysZone, SysErr)
  ...
  ]
```

will default arguments siz, zone, errRtn if missing or zero to 100, sysZone and SysErr respectively. Note that Bcpl will allow you to omit parameters in the middle of a parameter list by using "nil," but DefaultArgs has no way of knowing that you did this.

MoveBlock(dest, src, count) Uses BLT: for i = 0 to count-1 do dest!i = src!i.

SetBlock(dest, val, count) Uses BLKS: for i = 0 to count-1 do dest!i = val.

Zero(dest, count) Same as SetBlock(dest, 0, count).

BitBlt(bbt) Executes the BITBLT instruction with bbt in AC2.

Usc(a, b) Usc performs an unsigned compare of a and b and returns -1 if a<b, 0 if a=b, 1 if a>b.

Min(a, b), Max(a, b) Returns the minimum or maximum of two signed integers, which must differ by less than 2^15.

Umin(a, b), Umax(a, b) Returns the minimum or maximum of two unsigned integers.

DoubleAdd(a, b)	The parameters a and b each point to 2-word double-precision numbers. DoubleAdd does a_a+b. Note that subtraction can be achieved by adding the two's complement; the two's complement is the one's complement (logical negation) plus 1.
EnableInterrupts()	Enables Alto interrupt system.
DisableInterrupts()	Disables interrupt system. Returns true if interrupts were on.
StartIO(ac0)	Executes the SIO emulator instruction with its argument in ac0. Thus StartIO(#100000) will boot the Alto if it has an Ethernet interface.
Idle()	This procedure is called whenever the operating system is waiting for something to happen (e.g., a keyboard character to be struck, or a disk transfer to complete). The static lvIdle points to the operating-system copy of the procedure variable so that programmers may install their own idle procedures by executing "@lvIdle = MyIdle".
Timer(tv)	Reads the 32-bit millisecond timer into tv!0 and tv!1. Returns tv!1 as its value.
ReadCalendar(dv)	Reads the current date-and-time (32 bits, with a grain of 1 second) into dv!0 and dv!1. Returns dv as its value. (Subroutines for converting date-and-time into more useful formats for human consumption are available. See subroutine package documentation, under Time.)
SetCalendar(dv)	Sets the current date-and-time from dv!0 and dv!1. (Normally it should not be necessary to do this, as the time is set when the operating system is booted and has an invalid time. Thereafter, the timer facilities in the operating system maintain the current time.)
EnumerateFp(proc)	For every file pointer saved by the system (e.g., fpComCm, fpRemCm, etc.), call proc(fp).
CallSwat(s1, s2)	This function invokes an explicit "call" on Swat. Either of the arguments that appears to be a Bcpl string will be printed out by Swat.

3.10.1. Routines for Manipulating Bcpl Frames

The following routines ease massaging Bcpl frames for various clever purposes such as coroutine linkages. See section 4.7 for a description of the data structures involved.

FrameSize(proc)	Returns the size of the frame required by proc.
MyFrame()	Returns the address of the current frame.
CallersFrame(f)	Returns the address of the frame that "called" the frame f (if f is omitted, the current frame is used).
FramesCaller(f)	Returns the address to which the caller of frame f sent control, provided that he made the call with a normal instruction (jsrii, jsris). If error, returns 0.

CallFrame(f, a, b)	Sends control to frame f and links it back to this one (i.e., when f returns, the CallFrame call returns). a and b are optional arguments.
GotoFrame(f, a, b)	Like CallFrame, but does not plant a return link.
CoCall(a, b)	CallFrame(CallersFrame(), a, b)
CoReturn(a, b)	Like CoCall, but does not plant return link.
ReturnTo(label)	Returns to a given label in the frame of the caller.
GotoLabel(f, label, v)	Sends control to the specified label in the specified frame, and passes v in AC0.
RetryCall(a, b)	Repeats the call which appears to have given control to the caller with a and b as the first 2 arguments, and the other arguments unchanged. There are certain ways of calling functions which cannot be retried properly. In particular, the address of the procedure must be the value of a static or local variable; it cannot be computed. Thus "a>>proc(s, b)" cannot be retried, but "let pr=a>>proc; pr(s, b)" can be retried.
ReturnFrom(fnOrFrame, v)	Looks for a frame f which is either equal to fnOrFrame, or has FramesCaller(f) equal to fnOrFrame. It then cuts back the stack to f and simulates a return from f with v as the value. If error, it returns 0.

3.11. Subsystems and user programs

All subsystems and user programs are stored as "Run files", which normally have extension ".Run". Such a file is generated by Bldr and is given the name of the first binary file, unless some other name is specified for it. The format of an Alto run file is discussed in section 4.8 and in the Bcpl manual.

CallSubsys(S, pause [false], doReturn [false], userParams [0]) will read in a run file and send control to its starting address, where S is an open disk stream for the file, positioned at the beginning of the file. If pause is true, then CallSwat("Pause to Swat"); Ctrl-P starts the program. (doReturn will never be implemented, but would have allowed a return to the caller after the called subsystem "finished.") userParams is a pointer to a vector (length up to lUserParams) of parameters which will be passed to the called subystem. The parameters are formatted according to conventions given in SysDefs.D (structure UPE): each parameter is preceded by a word that specifies its type and the length of the block of parameters; a zero word terminates this list. When the Alto Executive invokes a program with CallSubsys, it passes in userParams an entry with type globalSwitches which contains a list of ASCII values of global switches supplied after the program name.

The open stream is used to load the program into Alto memory according to placement information included in the file. The stream is then closed; no other open streams are affected.

The program is started by a call to its starting address, which will normally be the first procedure of the first file given to Bldr. This procedure is passed three arguments. The first is the 32 word layout vector for the program, described in the Bcpl manual. The second is a pointer to a vector of parameters provided by the caller (the userParams argument to CallSubsys). The third is the "complete file address" (CFA) for a particular point in the file that was used to load the program. If no overlays are recorded in the Run file, this point is the end of file. If overlays are contained in the file, the CFA points to the first word of the first overlay section (this can be used as a hint in a call to OpenFile when loading overlays contained in the same file).

Subsystems conventionally take their arguments from a file called Com.Cm, which contains a string which normally is simply the contents of the command line which invoked the subsystem (see section 5). The subroutine package GP contains a procedure to facilitate reading this string according to the conventions by which it is normally formatted. This is not a standard routine but must be loaded with your program. (For more information on GP, see the "Alto Software Packages Manual.")

3.12. Finish -- Terminating Execution

When a program terminates operation, it "finishes," returns to the operating system and ultimately to the Executive. A program may finish in several ways:

Bcpl return	If the main procedure in the user program (the one invoked by CallSubsys) ever returns, the program finishes. Equivalent to OsFinish(fcOK).
Bcpl finish	If the "finish" construct is executed in a Bcpl program, it terminates. Equivalent to OsFinish(fcOK).
Bcpl abort	If the "abort" construct is executed in a Bcpl program, it terminates. Equivalent to OsFinish(fcAbort).
Swat abort	If, during program execution, the "left shift" key and the "Swat key" (lower-rightmost key on Alto I keyboards, upper-rightmost key on "ADL" Alto II keyboards) are depressed concurrently, the program is aborted. Similarly, if the <control>K ("kill") command is typed to Swat, the program is aborted. Both are equivalent to OsFinish(fcAbort).
OsFinish(fCode)	An explicit call to this function will also terminate execution. The value of fCode is saved in the static OsFinishCode, which may be examined by the Executive and the next program that it invokes. Values of fCode presently defined are: fcOK=0; fcAbort=1.

When a program finishes, the value of the finish code is first recorded. Then, if the value of the static UserFinishProc is non-zero, the call UserFinishProc(OsFinishCode) is performed before restoring the operating system state. This facility is useful for performing various clean-ups. (Note: To set UserFinishProc, it is necessary to execute @lvUserFinishProc = value.) In order to permit independent software packages to provide for cleanups, the convention is that each initialization procedure saves the present value of UserFinishProc and then replaces it with his procedure. This procedure will do the cleanups, restore UserFinishProc, and return:

```
// Initialization procedure
   ...
   static savedUFP
   savedUFP = @lvUserFinishProc
   @lvUserFinishProc = MyCleanUp
   ...

// The cleanup procedure
let MyCleanUp(code) be
   [
   ... cleanups here
   @lvUserFinishProc = savedUFP
   ]
```

Finally, control is returned to the operating system, which resets the interrupt system, updates the disk allocation table, and invokes the executive anew.

3.13. Junta

This section describes some procedures and conventions that can be used to permit exceptionally large programs to run on the Alto, and yet to return cleanly to the operating system. The basic idea is to let a program deactivate various operating system facilities, and thereby recover the memory devoted to the code and data used to implement the facilities. To this end, the system has been organized in a series of "levels:"

levBasic	Basic resident, including parity interrupt processing, time-of-day maintenance, the resident interface to the Swat debugger, and the initial processing for OsFinish. Important system state is saved here: EventVector, UserName, UserPassword, OsFinishCode. (Approximate size: 1000 words. This portion of the operating system is guaranteed not to extend below address 175000B.)
levBuffer	The system keyboard buffer (see section 4.6). (Approximate size: 100 words)
levFilePointers	File hints. This region contains "file pointers" for frequently referenced files. (Approximate size: 70 words)
levBcpl	Bcpl runtime routines. (Approximate size: 300 words)
levStatics	Storage for most of the system statics. (Approximate size: 300 words)
levBFSbase	Basic file system "base" functions, miscellaneous routines. (Approximate size: 1500 words)
levBFSwrite	Basic file system "write" functions, the disk descriptor (used to mark those pages on the disk which are already allocated), interface to the time-of-day clock. (Approximate size: 1850 words)
levAlloc	The Alloc storage allocation package. (Approximate size: 660 words)
levStreams	Disk stream procedures. (Approximate size: 2400 words)
levScan	Disk stream extension for overlapping disk transfers with computation. (Approximate size: 400 words)
levDirectory	Directory management procedures. (Approximate size: 1400 words)
levKeyboard	Standard keyboard handler. (Approximate size: 500 words)
levDisplay	Display driver (although the storage for the display bitmap and for the system font lie below). (Approximate size: 1600 words)
levMain	The "Main" operating system code, including utilities, CallSubsys, and the Junta procedure. (Approximate size: 1000 words)
-----	Below levMain, where the stack starts, the system free-storage pool is located. Here are kept stream data structures, the system font, and the system display bitmap. (Approximate size: 6000 words)

This table of levels corresponds to the order in which the objects are located in the Alto memory: levBasic is at the very top; the bottom of levMain is the highest location for the Bcpl stack.

The "Junta" function is responsible for de-activating these levels, thereby permitting the space to be reclaimed. When a program that has called Junta is ready to finish, it calls OsFinish in the normal way. OsFinish performs the "counter-junta," reading in portions of the operating system from the boot file and rebuilding the internal state of those levels that were previously de-activated, and then proceeds with the finish, calling the Executive, etc.

During the counter-junta process (which takes about 1/2 second), the display and interrupt system can continue to be active, provided that the code and storage they use lies below the address that is the value of OsFinishSafeAdr. This permits a token display to remain; also a keyboard handler can continue to sense key strokes and record characters in the system keyboard buffer.

Junta(levName, Proc)
: This function, which may be called only once before a "finish" or CounterJunta is done, de-activates all levels below levName. Thus levName specifies the name of the last level you wish to retain. (Manifest constants for the level names are in SysDefs.d.) It then sets the stack to a point just below the retained level, and calls Proc(), which should not return.

  The stack present at the time Junta is called is destroyed. The recommended procedure for saving data across a call to Junta is to locate the data below EndCode.

  A Junta always destroys the system free-storage pool and does not re-create it. Therefore, open streams, the system display and system font are all destroyed.

  It is the user's responsibility to take care not to call operating system procedures that lie in the region de-activated by the Junta. If in doubt, consult the file Sys.Bk, which documents the association between procedures and levels.

...finish...
: Any of the methods for terminating execution (section 3.12) automatically restores the full operating system.

CounterJunta(Proc)
: This function restores all de-activated sections of the operating system, and then calls Proc. The program stack present when CounterJunta was called is destroyed. This function is provided for those programs that do not wish to return to the operating system with a "finish," but may wish to do other processing (e.g., CallSubsys).

After calling Junta, many programmers will wish to restore some of the facilities that the Junta destroys, such as a free storage zone, a display stream, etc. Below is an example of how to go about this. Note that some thought is required because the operating system keeps a separate copy of statics from those referenced in your program. Thus when the OS defaults the third argument of CreateDisplayStream to sysFont, it uses the OS copy of sysFont, not the copy available to your program.

```
  ...
  Junta(levXXXXX, Proc)
  ...

let Proc() be
  [
  //Make a new sysZone:
  let v = vec 7035     // You can make it any size
  v = InitializeZone(v, 7035)
  @lvSysZone = v       // Patch the os's version of the static
```

```
sysZone = v          // Patch my program's version of the static

//Read in the system font again:
let s = OpenFileFromFp(fpSysFont)
let l = FileLength(s)/2
let f = Allocate(sysZone, l)
Resets(s); ReadBlock(s, f, l); Closes(s)
sysFont = f+2        // Patch my program's version of the static
                     // Note that because os's version is not patched,
                     //   I cannot call Ws or otherwise default dsp.

//Make a display stream:
dsp = CreateDisplayStream(6, Allocate(sysZone, 4000), 4000, sysFont)
ShowDisplayStream(dsp, DSalone)

...
```

3.14. Events

The operating system reserves a small communication region in which programs may record various things. The intended use for this region is the recording of events by one program that deserve attention by another. The Executive cooperates in invoking programs to deal with events posted in the communication region.

Events are recorded sequentially in a table pointed to by the static EventVector. The total length of the table, available as EventVector!-1, must not be exceeded by any program generating events. Each event entry (structure EVM; see SysDefs.d) contains a header that specifies the type and length of the entry (length is in words and includes header size); following the header comes type-specific data (eventData). A zero word terminates the event table.

At present, events are defined for:

eventBooted	The operating system has just been booted.
eventAboutToDie	The operating system is about to be flushed, probably to run a diagnostic.
eventInstall	The operating system is to be re-installed. (This event need only be used by the Executive "Install" command.)
eventRFC	A Request For Connection packet arrived. The event data is: Connection ID (2 words), RFC Destination Port (3 words), RFC Source Port (3 words) and Connection Port (3 words).
eventCallSubsys	When the next "finish" occurs, the system will try to execute the file whose name is given as a Bcpl string in the eventData block. If the eventData block has length 0, the system will invoke the copy of Ftp that is squirreled away inside Sys.Boot. Because a "finish" is performed right after the system is bootstrapped, it is possible to InLd Sys.Boot with a message that contains an eventCallSubsys, and thereby to invoke an arbitrary program. See the next section for a description of InLd.
eventInLd	Whenever the next "finish" occurs, the system will call InLd(eventData, eventData). This suggests that the first words of event data should be an FPRD for a file you wish to InLd.

If a program that generates an event has destroyed the event communication region, it is still possible to pass the event to the operating system. For example, if the memory diagnostic is running and an Ethernet connection request arrives, the mechanism can be used to load the operating system and pass the eventRFC message to it. The mechanism is described in the next section.

3.15. OutLd, InLd, BootFrom

Three functions are provided for dealing with "OutLd" files that record the entire state of the Alto machine. When the operating system is loaded with the "boot" button, such a file restores the machine state exactly as it was at the time of the Installation of the operating system. The Swat debugger also uses these facilities, saving the entire machine state on the file "Swatee" when a break is encountered, and restoring the Swat debugger state from the file "Swat."

In the discussion that follows, an FPRD structure is like a file pointer (FP), but the disk address is the Real disk address of the first page of Data in the file.

OutLd(FPRD, OutLdMessage) Saves the state of the machine on the file described by FPRD, which must exist and be at least 255 data pages long. Note that the state saved includes a PC inside OutLd. OutLd returns 0 after writing the file. Unless you know what you are doing, interrupts should be off when calling OutLd (otherwise, OutLd may save some parts of the machine state, such as the ActiveInterrupts word, that was pertinent to an interrupt in progress!).

Programmers should be warned to think carefully about the state that is being saved in an OutLd. For example, the operating system normally saves in memory some state associated with the default disk, sysDisk. If OutLd saves this state on a file, and the program is later resumed with InLd, the state will be incorrect. To be safe, state should be written out before calling OutLd (i.e., WriteDiskDescriptor()), and restored when OutLd returns (i.e., ReadDiskDescriptor()).

InLd(FPRD, InLdMessage) Copies the InLdMessage (length lInLdMessage) to a momentarily safe place and restores the machine state from the file described by FPRD, which must have been created by OutLd. Because the PC was in OutLd, OutLd again "returns," but this time with the value 1, and the InLdMessage has been copied into the OutLdMessage. Note: OutLd returns with interrupts disabled in this case.

If the operating system boot file is InLd'ed, the message is assumed to be a legal data structure for the EventVector, and is copied there.

BootFrom(FPRD) This function "boots" the Alto from the specified file. If it is applied to a file written by OutLd, the state of the machine is restored and OutLd "returns" 2 with interrupts disabled. (Note: The effect of this function differs from the effect of depressing the "boot" button. Unlike the boot button, the function in no way initializes the internal state of the Alto processor.)

Some programs (e.g., DMT) will need to know how to simulate InLd or BootFrom:

1. Turn off the display and disable interrupts.
2. Read the first data page of the boot file into memory locations 1, 2, ...#400. If you are loading the installed operating system, the first data page of the boot file is at real disk address 0.
3. Store the label block for the page just read into locations #402, #403, ...#411.

4. (This step applies only if simulating InLd.) Now let msa=rv 2. This points to a location where a brief message can be stored. Set msa!0=1. Then for i=0 to lInLdMessage-1 do msa!(i+1) = PrototypeEventVector!i.
5. Jump to location 3, never to return.

4. Data structures

This section describes the data structures used by the operating system that may be required by users.

4.1. Reserved Memory Locations

The Alto Hardware Manual describes addresses reserved for various purposes. The file AltoDefs.d distributed with the OS declares most of these as manifest constants.

4.2. Streams

The standard data structures for streams are given in the DiskStreams package file "Streams.d". Documentation for the streams package includes a description.

4.3. Disk files

The structure of the Alto file system is described in documentation for the Alto file system (Disks). This includes a description of files, disk formats, directory formats, and the format of the disk descriptor. Bcpl declarations for these objects may be found in the file AltoFileSys.d.

4.4. Display

The data structures used to drive the Alto display are described in the Alto Hardware Manual. The font format for the Alto (.AL format) is also described there. Note that a font pointer such as the one passed to CreateDisplayStream points to the third word of an AL font.

4.5. Zones

A program that wishes to create an operating-system object and retain control over the allocation of storage to the object may pass a "zone" to the operating system function that needs space (e.g., CreateDiskStream). A zone is simply a pointer "zone" to a structure ZN (see SysDefs.d), with zone>>ZN.Allocate containing the address of the allocation procedure (called by (zone>>ZN.Allocate)(zone, lengthRequested)) and zone>>ZN.Free containing the address of the free procedure (called by (zone>>ZN.Free)(zone, block)). The zones created by the Alloc allocator package obey these conventions.

The zone provided by the operating system is saved in the static sysZone. The user may replace the system zone by executing @lvSysZone = value. Subsequent free-storage requirements for the operating system will be addressed to this zone. The system zone is restored when the user program terminates. Warning: The operating system keeps various (and undocumented) information in the system zone, and is unwilling to have the zone changed out from under it. The normal use of lvSysZone is to change the value of sysZone immediately after a call to Junta (which clears away sysZone). If you wish to create disk streams and preserve them across a call to Junta, pass your own zone as an argument to OpenFile.

4.6. Operating System Status Information

A good deal of information is retained in memory that describes the state of the Alto. Much of this information is of relevance to programmers, and is contained in some static scalars:

OsVersion	The version number of the operating system. This number is incremented with each new release of the operating system, incorporating changes however minor.
OsVersionCompatible	The lowest operating system version number believed to be compatible with the present system.
UserName	This static points to a Bcpl-format string that is the user's last name. It is initialized when the operating system is installed on the disk. The maximum length (in words) that the UserName may occupy is recorded in UserName!-1.
UserPassword	This static points to a Bcpl-format string that is the user's password, typed to the Executive Login command. The maximum length (in words) that the UserPassword may occupy is recorded in UserPassword!-1.
SerialNumber	The serial number of the Alto you are on. This static has proved troublesome, because it is easy to forget that this too will be saved by OutLd, and can confuse Ethernet code when it suddenly springs to life months later on a different host half way around the world. Its use is discouraged.
AltoVersion	This static contains the result of executing the VERS instruction. This static has proven troublesome for the same reasons as SerialNumber. Its use is discouraged.
sysDisk	A pointer to the DSK structure, described in Disks.d, which describes the "disk" to be used for standard operating system use. This structure is actually of the format BFSDSK, and contains a copy of the DiskDescriptor data structure. The static diskKd points to this structure alone (structure KD; see AltoFileSys.d). The storage for sysDisk is in levBFSwrite; if you Junta to levBFSbase, you will need to manufacture a new sysDisk structure, by loading and calling BFSInit in your own program.
lvSysErr	This static points to the operating-system copy of the static that contains the address of the error procedure. If you wish to replace SysErr, it suffices to say @lvSysErr=Replacement. Note that some procedures may have already copied the value of SysErr (e.g., when a stream is created, the value of SysErr is copied into the ST.error field in most cases).
lvParitySweepCount	This static contains the address of the highest memory location examined when sweeping memory looking for parity errors. If no parity checking is desired, set @lvParitySweepCount = 0.
lvParityPhantomEnable	This static points to a flag that determines whether phantom parity errors will invoke Swat (a phantom parity error results from a parity interrupt that can find no bad locations in memory). @lvParityPhantomEnable=0 will disable phantom reporting.

ErrorLogAddress — This static points to a network address of a spot where error reports (for such things as parity errors) should be sent. The structure is a "port," as defined in Pup documentation.

ClockSecond — This static points to a double-precision integer that gives the count of number of RCLK ticks (when RCLK is viewed as returning a 32-bit number) in a second. This number is used for keeping time, and is nominally 1680000. If timekeeping is extremely critical, you may wish to calibrate your Alto and change this number.

File Hints — The operating system maintains file pointers for several commonly-used files. Using these hints in conjunction with OpenFile will substantially speed the process of opening streams. The files and file pointers are:

SysDir	fpSysDir
SysBoot	fpSysBoot
DiskDescriptor	fpDiskDescriptor
User.Cm	fpUserCm
Com.Cm	fpComCm
Rem.Cm	fpRemCm
Executive.Run	fpExecutive
SysFont.Al	fpSysFont

Keyboard Buffer — Although the system keyboard buffer is normally managed by the keyboard handler provided in the system, some programs may want to operate on it themselves. The most important instance of this is when a program that has done a Junta is finishing: if the program keeps its keyboard handler enabled, any characters typed during the counter-junta can still be recorded in the system buffer, and thus detected by the first program to run (usually the Executive).

The static OsBuffer points to a structure OsBUF (see SysDefs.d) that controls access to the buffer:

OsBuffer>>OsBUF.First	First address of the ring buffer
OsBuffer>>OsBUF.Last	Last address of the ring buffer+1
OsBuffer>>OsBUF.In	"Input" pointer (place to put next item)
OsBuffer>>OsBUF.Out	"Output" pointer (place to take next item)

The following code can be executed with interrupts on or off to deal with the buffer:

```
GetItem() = valof      //Returns 0 if none there!
   [
   if OsBuffer>>OsBUF.In eq OsBuffer>>OsBUF.Out then resultis 0
   let newOut = OsBuffer>>OsBUF.Out+1
   if newOut eq OsBuffer>>OsBUF.Last then newOut = OsBuffer>>OsBUF.First
   let result = @(OsBuffer>>OsBUF.Out)
   OsBuffer>>OsBUF.Out = newOut
   resultis result
   ]

PutItem(i) = valof      //Returns 0 if buffer full already
   [
   let newIn = OsBuffer>>OsBUF.In+1
   if newIn eq OsBuffer>>OsBUF.Last then newIn = OsBuffer>>OsBUF.First
   if newIn eq OsBuffer>>OsBUF.Out then resultis 0
```

```
        @(OsBuffer>>OsBUF.In) = i
        OsBuffer>>OsBUF.In = newIn
        resultis -1
        ]

    GetItemCount() = valof   //Returns count of items in buffer
        [
        let c = OsBuffer>>OsBUF.In-OsBuffer>>OsBUF.Out
        if c ls 0 then c = c+OsBuffer>>OsBUF.Last-OsBuffer>>OsBUF.First
        resultis c
        ]

    ResetItemBuffer() be    //Set buffer to empty
        [
        OsBuffer>>OsBUF.In = OsBuffer>>OsBUF.First
        OsBuffer>>OsBUF.Out = OsBuffer>>OsBUF.First
        ]
```

#176777 — This location, the last in memory, points to the beginning of the area used to save statics for levBasic through levBcpl. The file Sys.Bk documents offsets from this number where the various statics will be found.

4.7. Swat

The operating system contains an interface to the Swat debugger (described in the "Alto Subsystems" manual). This interface uses OutLd to save the state of the machine on the file "Swatee," and InLd to restore the state of the machine from the file "Swat," which contains the saved state of the debugger itself. The inverse process is used to proceed from an interrupt or breakpoint. Two aspects of the Swat interface are of interest to programmers:

lvAbortFlag — If @lvAbortFlag is zero, holding down the <left-shift> and <B3> keys will simulate the call OsFinish(fcAbort), thus terminating execution of the running program. In critical sections, setting @lvAbortFlag to a non-zero value will disable aborts. The standard convention is to increment @lvAbortFlag when entering such a section and to decrement it when exiting. This permits separate software modules to use the feature concurrently.

lvSwatContextProc — Although Swat saves and restores the state of the standard Alto I/O devices, it has no way to know about special devices attached to the machine. The programmer may arrange that a peice of code will be called whenever Swat is trying to turn off I/O preparatory to calling OutLd, or trying to restart I/O after an InLd. If the programmer does @lvSwatContextProc=DLSProc, Swat will execute DLSProc(0) when turning off I/O, and DLSProc(-1) when turning it on. Since Swat can be invoked at any time, the Swat context procedure must be written in machine language and must not assume anything about the state of the machine or any data structures (in particular the Bcpl stack may be in an inconsistant state).

4.8. The Bcpl stack

The Bcpl compiler determines the format of a frame and the calling convention. The strategy for allocating stack frames, however, is determined by the operating system. We begin by describing the compiler conventions, which are useful to know for writing machine-language routines.

A procedure call: p(a1, a2, ...), is implemented in the following way. The first two actual arguments are put into AC0 and AC1 (AC2 always contains the address of the current frame, except during a call or return). If there are exactly three actual arguments, the third is put into F.extraArguments. If there are more than three, the frame-relative address of a vector of their values is put there (except for the first two), so that the value of the i-th argument (counting from 1) is frame>>F.extraArguments!(frame+i). Once the arguments are set up, code to transfer control is generated which puts the old PC into AC3 and sets the PC to p. At this point, AC3!0 will be the number of actual arguments, and the PC should be set to AC3+1 to return control to the point following the call.

A procedure declaration: let p(f1, f2, ...) be ..., declares p as a static whose value after loading will be the address of the instruction to which control goes when p is called. The first four instructions of a procedure have a standard form:

```
STA 3 1,2          ; AC2>>F.savedPC _ AC3
JSR @GETFRAME
<number of words needed for this procedure's frame>
JSR @STOREARGS
```

The Bcpl runtime routine GETFRAME allocates storage for the new frame, NF, saves AC2 in NF>>F.callersFrame field, sets AC2 to NF, and stores the values of AC0 and AC1 (the first two arguments) at NF>>F.formals ^0 and 1. If there are exactly three actual arguments, it stores the third one also, at NF>>F.formals ^2. Then, if there are three or fewer actual arguments, it returns to L+3, otherwise it returns to L+2 with the address of the vector of extra arguments in AC1; at this point a JSR @STOREARGS will copy the rest of the arguments. In both cases, the number of actual arguments is in AC0, and this is still true after a call of STOREARGS. A Bcpl procedure returns, with the result, if any, in AC0, by doing:

```
JMP @RETURN
```

to a runtime routine which simply does:

```
LDA 2 0,2          ; AC2_AC2>>F.callersFrame
LDA 3 1,2          ; PC_AC2>>F.savedPC+1
JMP 1,3
```

The information above is a (hopefully) complete description of the interface between a Bcpl routine and the outside world (except for some additional runtime stuff which is supplied by the operating system). Note that it is OK to use the caller's F.Temp and F.extraArguments in a machine-language routine which doesn't get its own frame, and of course it is OK to save the PC in the caller's F.savedPC.

The operating system currently allocates stack space contiguously and grows the stack down. To allocate a new frame of size S, it simply computes NF=AC2-S-2 and checks to see whether NF > EndCode. If not, there is a fatal error (Swat breakpoint at finish+1); if so, NF becomes the new frame. (Note: the "-2" in the computation is an unfortunate historical artifact.)

4.9. Run files

The format of a file produced by Bldr to be executed by CallSubsys is described by the structure definition SV in BCPLFiles.d. Consult the Bcpl manual (section on Loading) for interpretations of the various fields and the handling of overlays.

5. The Executive

The Alto Executive is itself a subsystem and lives on the file Executive.Run; if you don't like it, you can write your own. It is currently invoked from scratch after the operating system is booted, and whenever a subsystem returns. The Executive is fully documented in the "Alto Subsystems" manual.

6. Operating Procedures

6.1. Installing the operating system

The "Install" command causes the operating system to execute special code which completely initializes the system. The options of the install procedure are controlled by prompts. Installation is needed:

- When a new version of the operating system is distributed. New versions are called "NewOS.boot" (or "NewOsV.boot", the variant that supports the file version numbering facility). You should transfer NewOS.boot to your disk and install it by saying "Install NewOs.Boot". It will ask you several questions which determine it's configuration on your disk ("SysGen", if you will parden the expression) and finally the Executive will be invoked. The newly configured OS writes itself on the file Sys.boot, so you can delete NewOS.boot after installing.

- When you wish to ERASE a disk completely and re-initialize it. This option pauses to let you insert the disk pack you want initialized. This "new disk" function is invoked by answering affirmatively the question "Do you want to ERASE a disk before installing?" after answering affirmatively that you want the "Long installation dialogue". See also the NEWDISK section of the Alto Subsystems Manual.

- When you wish to change the "user name" or "disk name" parameters of the operating system. The install procedure will prompt for these strings. It is also possible to specify a disk password that will be checked whenever the operating system is booted.

- When you wish to enable the "multiple version" feature of the file system. (Because few programs presently cope with all the subtleties of this feature, it is wise to leave it disabled.)

- When you wish to extend a file system. Basic disks are often kept on Interim File Systems from which users can copy them with CopyDisk. They are usually configured for a single Diablo model 31 disk. If your machine has more disk space, you can extend the file system by answering "Yes" to the question "Do you want to extend this file system?" (this is also part of the "long installation dialog").

6.2. How to get out of trouble

It occasionally happens that a disk will not boot, or something runs awry during the booting process. In this case, the following steps should be considered:

1. Run the Scavenger. This can be done in two ways:

   Place a good disk in the Alto, and invoke the Scavenger. When it asks if you wish to change disks, respond affirmatively, put the damaged disk in the machine and proceed when the drive becomes ready.

   If you have network access to a "boot server", hold down the <BS> and <'> keys and push the boot button. Continue to hold down <'> until a tiny square appears in the middle of the screen. You should now be talking to the Network Executive; type Scavenger<cr>.

When the Scavenger finishes, the attempt to invoke the Executive may fail because Scavenger was invoked from another disk. Try booting. If unsuccessful, go on to step 2.

2. Use Ftp to get fresh copies of SysFont.al and Executive.Run. Again, this can be done in two ways:

   Place a good disk in the machine and invoke Ftp. After it is initialized, change disks, wait for the damaged one to become ready, and type the necessary Ftp commands to retrieve the files.

   Invoke Ftp via the Network Executive as in step 1.

   Now try booting. If unsuccessful, go to step 3.

3. Install the OS. You guessed it; this can be done in two ways:

   Place a good disk in the Alto and type "Install." When asked for your name, place the damaged disk in the machine, wait for the drive to become ready, and proceed.

   Invoke the "NewOS" via the Network Executive. You will be asked: "Do you want to INSTALL this operating system?"

6.3. File Name Conventions

Various conventions have been established for Alto file names. The conventions are intended to be helpful, not authorative.

1. All files relating to a subsystem "Whiz" should have file names of the form "Whiz.xxx", i.e. typing "Whiz.*" to the Executive should list them all, delete them all, etc. Example: Bcpl.Run, Bcpl.Syms, etc.

2. File extensions are of preference chosen to be <u>language</u> extensions, i.e. they specify the language in which they are written. The present set is:

Bcpl	Bcpl source code
Mu	Micro-code source
Asm	Assembler source code
Mesa	Mesa source code
Help	A help file for the system given in the name
Cm	A command file for the Alto Executive

3. File extensions are otherwise chosen to reflect the <u>format</u> of the file. The present set is:

Bravo	Text file with Bravo format codes
Run	Executable file produced by Bldr
Image	Executable file produced by Mesa
Al	Alto format font file
Boot	A file that can be booted
Br	Bcpl relocatable binary file
Syms	Bldr symbol table output
BCD	Mesa object code
Dm	File produced by the Dump command, read by the Load command
Ts	Text file containing a transcript
Disk	disk image CopyDisk format
..	

6.4. Miscellaneous information

The key in the lower right corner of the keyboard on a Microswitch keyboard (<blank-bottom>) or in the upper right on an ADL keyboard (FR1) is called the Swat key. If you press it, as well as the <ctrl> and <left-shift> keys, the Swat debugger will be invoked. If you do this by mistake, <ctrl>P will resume your program without interfering with its execution, and <ctrl>K will abort your program.

You can force an abort at any time by depressing the Swat key together with the <left-shift> key.

In order for the operating system to run properly, the following files should be on your disk (those marked * are optional):

SysDir	System directory.
DiskDescriptor	Disk allocation table.
SysFont.Al	System display font.
Executive.Run	Executive (command processor).
Sys.Boot	Boot-file containing the operating system.
Sys.Errors	* Error messages file.
Swat	* Debugger program, created by running InstallSwat.
Swatee	Debugging file essential to Swat.

(Note: If you wish to change the font used by the operating system, it suffices to copy a new font to SysFont.Al and boot the system.)

If you intend to write programs that use the operating system facilities, you will want some additional files:

Sys.Bk	Required by Bldr to load programs that reference operating system functions. This file also shows which functions are implemented in which levels and the names of source files for the code.
SysDefs.d	Definitions of standard system objects. You will probably want to "get" this file in Bcpl compilations that use operating system functions extensively.
Streams.d	Data structure definitions relating to streams.
AltoFileSys.d	Data structure definitions relating to files.
Disks.d	* Data structure definitions relating to the "disk" object.
AltoDefs.d	Definitions of places and things peculiar to an Alto.
BcplFiles.d	* Definitions of the formats of Bcpl-related files.

Name	Opcode	Address	Function
CYCLE	60000	C	AC0_AC0 lcy (if C ne 0 then C else AC1); smashes AC1
JSRII	64400	D	AC3_PC+1; PC_rv (rv (PC+D))
JSRIS	65000	D	AC3_PC+1; PC_rv (rv (AC2+D))
CONVERT	67000	D	character scan conversion
DIR	61000	-	disable interrupts
EIR	61001	-	enable interrupts
BRI	61002	-	PC_interruptedPC; EIR
RCLK	61003	-	AC0_16 msb of clock (from realTimeClock); AC1_ 10 lsb of clock * #100 + 6 bits of garbage; resolution is 38.08 us.
SIO	61004	-	start I/O
BLT	61005	-	Block transfer of -AC3 words; AC0=address of first source word-1; AC1=address of <u>last</u> destination word; AC0 and AC3 are updated during the instruction
BLKS	61006	-	Block store of -AC3 words; AC0=data to be stored; AC1=address of <u>last</u> destination word; AC3 is updated during the instruction
SIT	61007	-	start interval timer. For an interrupt when the time is timerInterruptTime, AC0 should be 1 when this instruction is executed
JMPRAM	61010	-	Emulator microcode PC_AC1 in control RAM
RDRAM	61011	-	AC0_(if AC1[4] then RAM else ROM)!AC1 (left half if AC1[5], right half otherwise)
WRTRAM	61012	-	RAM!AC1_(AC0,AC3)
DIRS	61013	-	* Disable interrupts and skip if interrupts were on
VERS	61014	-	* AC0_((EngineeringNumber-1)*16 +BuildNumber)*256 +MicrocodeVersion
DREAD	61015	-	** AC0_rv(AC3); AC1_rv(AC3 xor 1)
DWRITE	61016	-	** rv(AC3)_AC0; rv(AC3+1)_AC1
DEXCH	61017	-	** t_rv(AC3); rv(AC3)_AC0; AC0_t; t_rv(AC3+1); rv(AC3+1)_AC1; AC1_t
MUL	61020	-	Same as NOVA MUL: AC0,1_AC2*AC1+AC0
DIV	61021	-	Similar to NOVA DIV: AC1_AC0,1/AC2; AC0 has remainder. DIV (unlike NOVA version) skips the next instruction if no overflow occurs.
BITBLT	61024	-	* character scan conversion of bit-map manipulation

Notes: Address: C=bits 12-15; D=bits 8-15; -=no address
variables in function descriptions are machine registers or page 1 locations
* indicates available only in "new" microcode (SIO leaves AC0[0]=0)
** indicates available only on Alto II

Table 2.1: New instructions in Alto emulator
(see Alto Hardware Manual for more details)

Device	Diablo 31	Diablo 44	
Number of drives/Alto	1 or 2	1	
Number of packs	1 removable	1 removable 1 fixed	
Number of cylinders	203	406	
Tracks/cylinder/pack	2	2	
Sectors/track	12	12	
Words/sector	2 header 8 label 256 data	same	
Data words/track	3072	3072	
Sectors/pack	4872	9744	
Rotation time	40	25	ms
Seek time (approx.)	15+8.6*sqrt(dt)	8+3*sqrt(dt)	ms
min-avg-max	15-70-135	8-30-68	ms
Average access to 1 megabyte	80	32 (both packs)	ms
Transfer rates:			
peak-avg	1.6-1.22	2.5-1.9	MHz
peak-avg	10.2-13	6.7-8	us-word
per sector	3.3	2.1	ms
for full display	.46	.27	sec
for big memory	1.03	.6	sec
whole drive	19.3	44 (both packs)	sec

Table 2.2: Properties of Alto disks

LastMemLoc	Last memory location
...	
StartSystem	Base of system
...	
StackBase	Root of stack; stack extends downward from here
...	
StackEnd	Top of stack, which grows down
...	
EndCode	End of user program+1
...	This space contains user code and statics, loaded as specified by the arguments to Bldr. Default is to start at StartCodeArea and load statics into the first 400 words, and code starting at StartCodeArea+400. See Bcpl manual.
StartCodeArea	Start of user program area
...	
400-777	Page 1: machine-dependent stuff (see Alto Hardware Manual)
...	
300-377	Bcpl runtime page 0
...	
20-277	User page 0
...	
0-17	Unused

Table 3.1: Memory layout (all numbers octal); see section 3.6

LastMemLoc	The operating system described in this document runs on 64K Altos; this location is 176777.
StackEnd	The address of the frame in which the current procedure is executing is computed by the MyFrame procedure; alternatively, compute lv (first argument of current procedure) -4
EndCode	Rv(335)
StartCodeArea	User code may start at any address > 777.

Table 3.2: Values of symbolic locations in Table 3.1
(all numbers octal)

Operating System Change History

This file contains an inverse chronological listing of changes to the Alto operating system.

The "normal way" to install a new operating system is to retrieve a copy of the files NewOS.Boot, Sys.Syms, Sys.Errors and Sys.Bk that are being distributed. Say "Install NewOS.boot" to the Exec, answer the configuration questions and then delete NewOs.Boot.

Version 19/16 -- December 15, 1980

Additions: The major addition is that you can now erase a disk and format it to use 14 sectors per cylinder on D0s and Dorados. It is not possible to extend a 12 sector file system to 14 sectors "in place"; you must save your files, erase the disk and restore them.

Changes: [BFSInit] The OS refuses to boot when only one disk of a double disk file system is spinning. It can also detect certain other blunders like DP1 containing a single disk file system rather than the second half of the filesystem starting on DP0. It is not possible to detect all bad cases. [KeyStreams] the static kbTransitionTable is not exported to users who wish to modify the OS's treatment of the keyboard. [DspStreams] it used to be that character codes below 40b unconditionally called the stream scroll procedure. Now, if the character has a non-zero width or height it is displayed. Only characters with zero width and height (CR and LF in particular) call scroll.

Version 18/16 -- May 5, 1980

Additions: The major addition is that you can now extend a file system by reinstalling the OS. A single model 31 file system can be extended to a double model 31, a single model 44 or a double model 44, and a single model 44 can be extended to a double model 44. This is accomplished by a subdialog of the 'long installation dialog'.

Changes: [Calendar] D0s and Dorados now use Alto I clock format. [Dirs] A bug in the 'CompareFn' feature has been fixed. [BFS] 'return on check error' is handled better. [InOutLd] Disk error recovery during InLd and OutLd has been improved. [DiskStreams] A bug in FilePos, introduced in OS17 and responsible for problems with long files in FTP, has been fixed. CleanupDiskStream now does the proper thing if a file is extended to a multiple of the page size and then trimmed back by less than a page. [DisplayStreams] EraseBits is much faster now because it uses BitBlt. [BfsMl] BitBlt calls Swat if the BBT starts at an odd address.

Version 17/16 -- September 9, 1979

The most significant improvements are that the DSK object has been extended to permit disk-independent operation at the DoDiskCommand/GetCb level; procedures have been added to scan a disk stream at full disk speed; and the directory lookup procedures have been modified to take advantage of these facilities and thereby improve performance substantially. To make way for these improvements, all support for file version numbers (a little-used feature) has been removed.

Incompatibilities are confined to those programs that create DSK objects, since several of the OS routines now expect to be passed the extended versions. Programs that include the TFS must be reloaded with the latest release of TFS; they will then run under OS 17 or OS 16. Programs that include BFSInit must be reloaded with the OS 17 version of BFSInit; they will then not work under previous OS releases. Of the standard Alto subsystems, FTP falls into the first category and Neptune in the second.

In the DSK object, the fields fpDiskDescriptor, driveNumber, retryCount, and totalErrors have moved, and fpSysLog has been deleted; it is believed that no existing programs are affected by this.

Additions: [BFS] the DSK object is extended to include generic procedures InitializeDiskCBZ, DoDiskCommand, GetDiskCb, and CloseDisk, and constants lengthCB and lengthCBZ. The CBZ structure is now public, and is defined in Disks.d and documented in the "Disks and BFS" description. InitializeDiskCBZ defaults its errorRtn argument. DoDiskCommand has an optional nextCb argument. DefaultBfsErrorRtn and BfsNonEx are exported in Sys.bk, so user programs can load BFSInit. The BFS can now operate in any of the file system partitions available on the large disks of Dorados and D0s. An optional hintLastPage argument to ActOnDiskPages, WriteDiskPages, and DeleteDiskPages has been added. New procedures include Min, Max, Umin, Umax, and Call10 through Call15.

[Disk streams] A DiskStreamsScan level has been added, containing the procedures InitScanStream, GetScanStreamBuffer, and FinishScanStream; these support overlapped reads at full disk speed.

[Keyboard] Shift-LF generates Ascii 140B -- accent grave.

Deletions: The remaining vestiges of the Sys.Log code are gone. BFSSetStartingVDA removed -- use ReleaseDiskPage(disk, AssignDiskPage(disk, desiredVDA-1)). All support for version numbers has been removed from the standard release; an alternate release (NewOsV.boot) is available in which the version number facility has been retained, but it does not benefit from the improved directory lookup performance, it is somewhat larger, and it may not be supported in the future.

Changes: levBasic is now guaranteed to be at 175000B or higher, for the benefit of Mesa and Smalltalk. ReleaseDiskPage doesn't increment the page count if the page released is already free. The BFS now retries data-late errors indefinitely. The BFS cleanup routine is now called with three arguments. The DiskDescriptor file is now allocated next to SysDir rather than in the middle of the disk as it was in OS 16. The old write date is not restored to a directory file (directory bit on in serial number) if the file is opened for writing but never dirtied. A number of bugs in the disk streams code have been fixed that prevented manipulation of files greater than 32767 pages long. Directory operations (OpenFile, DeleteFile, etc.) now search the directory at essentially full disk speed. Booting has been speeded up somewhat. The OS uses and maintains disk shape information as a DSHAPE file property in the leader page of SysDir.

Version 16/16 -- February 19, 1979

This version contains many internal changes but few external ones. Even though it is technically incompatible with previous releases (OS 16/16 rather than OS 16/5), most programs are not affected. There are three major changes: 1) backward compatibility for the "old" OS has been removed, 2) the disk bit table is now paged rather than occupying a fixed area in memory, and 3) the interface between Swat and the OS changed - Swat.25 is required.

Additions: the BitBlt instruction is accessible from Bcpl and a structure definition for a BitBlt table was added to AltoDefs.d. More of the page 1 and I/O area location names were added to AltoDefs.d. A new declaration file, BcplFiles.d, was created and the Bcpl file format definitions were moved there from SysDefs.d. The OS corrects parity in extended memory banks during booting. The "new" file date standard is implemented. The DDMgr object operations were added to Calls.asm.

Deletions: the compatiblity package has been removed. All of the commonly used subsystems which depended on it have been updated. They are: Asm, RamLoad, CleanDir, EDP, and Scavenger. If you keep any of these on your disk, you should get new copies from the <Alto> directory. fpSysLog, fpSysTs, fpWorkingDir, faSysLog, and nameWorkingDir went away.

Reorganiztions: the BFS was extensively reorganized to bring it into sync with the TFS. The code for creating a virgin file system and creating a DSK object has been disentangled from OS initialization. The Bcpl frame-munging code was split out of BFSML.asm and put into a new file: BcplTricks.asm. Initialization for the keyboard was moved from the OS initializtion modules into KeyStreamsB.bcpl, making it self-contained. Parity Error handling, Calendar clock update, Swat interface, and InOutLd were split into separate modules.

Changes: DisableInterrupts returns true if interrupts were on. The VERS and DCB structure were moved into AltoDefs.d. The names of many OS modules changed. The long installation dialog permits more precise control over the handling of memory errors. The erase disk dialog permits you to create an extra big directory. The interface to Swat has changed - Swat.25 is the new version.

Version 15/5 -- March 15, 1978

Fixed a bug in the file date code; introduced another bug in the same code.

Version 14/5 -- March 1, 1978

Additions: ReadCalendar and SetCalendar - analogus to DayTime and SetDaytime only they conform to the new time standard. DayTime and SetDaytime will continue work correctly until April 30, 1978. A new declaration file, AltoDefs.d was created; some things were moved there from SysDefs.d. Definitions of the format of .BB (overlay), and .Syms files were added to SysDefs.d. This OS has room for a 'big' bittable - a special OS version is not required.

Deletions: The system log was de-implemented. LogOpen, LogClose, and MakeLogEntry are now Noops. They will be removed when an incompatible OS is next released.

Reorganizations: Noop, TruePredicate and FalsePredicate were moved from StreamsML.asm to BFSML.asm (up a few Junta levels). Fast streams were split out of disk streams: FastStreamsB.bcpl and FastSteamsA.asm. Streams.bcpl was split into 3 files: DiskStreams.bcpl, DiskStreamsMain.bcpl, and DiskStreamsAux.bcpl; StreamsML.asm disappeared.

Changes: A bug in ReturnFrom was fixed (this only matters if you use the microcode version of the frame allocator). TruePredicate now returns -1 (it used to return 1). If the unrecoverable disk error routine in the BFS returns, the cleanup procedure is called and things plunge on. OpenFile with a filename containing a non-existant directory now returns 0 instead of calling Swat. The Diablo printer bits (0-7) are now ignored by the keyboard interrupt routine.

Version 13/5 -- May 16, 1977

Additions: ParseFileName (a lower level directory function) was made available to users.

Changes: Minor, yea insignificant bugs fixed.

Version 12/5 -- March 20, 1977

Additions: ClockSecond. Location 613b is now reserved to indicate to RAM microcode what sort of Alto we are on: 0 implies Alto I; -1 implies Alto II.

Changes: Time-keeping accuracy improved slightly. BFS is now reentrant--you may have several independent disk activities going concurrently (this will make CopyDisk more reliable).

Version 11/5 -- January 9, 1977

Additions: eventInLd and eventCallSubsys processing added. Also now possible to install the operating system with logging disabled.

Changes: Booting process somewhat more robust. Several changes to improve diagnostic information about parity errors provided by Swat. Improved password protection. Alto II fixes in parity and timer routines.

Version 10/5 -- November 2, 1976

Changes: A nasty bug in the disk routines was uncovered and fixed. It was responsible for occasionally garbaged files.

Version 9/5 -- September 25, 1976

Additions: verNewAlways option to OpenFile; changeSerial entry on file leader pages.

Changes: Various bugs relating to keeping file version numbers were fixed.

Version 8/5 -- August 28, 1976

Changes: Several bugs in parity error detection and reporting were removed.

Version 7/5 -- August 10, 1976

Additions: The Idle procedure and corresponding static lvIdle; lvParityPhantomEnable global static; more installation options.

Minor changes: Two bugs in PositionPage are fixed -- one permitted read-only files to be accidently lengthened.

Version 6/5 -- July 8, 1976

Additions: (1) Several global statics have been added: AltoVersion (code for machine, build and microcode versions), ErrorLogAddress (Ethernet address to report hardware errors), #176777 points to the global statics.

(2) The format of Sys.Boot has been altered slightly so that Altos may be booted over the Ethernet.

Version 5/5 -- April 28, 1976

How to get it: Because version 5 introduces some incompatibilities, it is essential that several subsystems be updated: (1) get a new Executive and Bravo 5.5 or later (these will run under version 4 or version 5 of the operating system); (2) get Sys.Bk, Sys.Syms, Sys.Boot (under another name, e.g. NewOs.Boot); (3) install your new system; (4) get a new version of DDS, which depends on version 5 of the operating system; (5) get a new InstallSwat.Run and invoke it; (6) if you are a programmer, be sure to get new copies of all definitions files (e.g. AltoFileSys.d).

Incompatibilities: (1) Most calling sequences and subroutine names for the "Bfs" routines have changed. These changes were made in order to introduce the concept of a "disk" object, so that standard OS stream and directory functions could be applied to non-standard disks (e.g., the Trident T80). The static lvDiskKd has been removed.

(2) The "disk address" returned as part of a CFA or FA is now a virtual disk address. The routine RealDiskDA can be used to convert it to a physical disk address if desired.

Minor changes: (1) The handling of the UserFinishProc has changed. The recommended procedure for such procedures is to simply return from a finish procedure, not to call OsFinish again.

(2) Several bugs in the streams package are fixed, e.g. ReadBlock applied to a file with 511 bytes in the last data page did not work correctly.

(3) The "new disk" refreshing procedure has been changed to use the new FTP; it is now mandatory that this file be present on your disk when you attempt to make a brand new disk.

(4) It is now possible to change disk packs during the Install sequence; simply change packs when some question is asked of you (exception: if you are creating a "new disk," do not change packs until told to do so).

(5) The log functions have been made much more robust. It is now possible to delete Sys.Log and continue operations.

(6) Numerous bugs in ReturnFrom and FramesCaller are fixed.

(7) The default number of file versions to keep is now stored in the DiskDescriptor.

(8) Wns has been changed to allow both signed and unsigned number conversion.

(9) The arguments to DeleteFile have changed slightly (only if you pass more than 2 arguments to it).

(10) The introduction of the "disk" object has added some statics: sysDisk, some functions: KsGetDisk, LnPageSize, and optional "disk" arguments to disk stream opening functions.

Operating System Software Packages

Several of the modules of the operating system are also available as software packages in case the programmer wishes to include them in overlays, or modify them, etc. The sources are in <AltoSource>OSSources.dm, and the binaries are in <Alto>OSBrs.dm. You are urged to get listings and ponder them since proper use of these procedures in a foreign context may require some modifications, and will certainly require some understanding. The BootBase package, in the BuildBoot documentation in the Subsystems manual, offers configurations of these packages that permit making most any subsystem into a boot file without souce level changes.

Utilities. The file OsUtils.Bcpl contains several of the utility procedures located in levMain: Wss, Ws, Wl, Wns, Wos, Wo, GetFixed, FreeFixed, FixedLeft, SetEndCode. The procedure GetFixedInit must be called to initialize the GetFixed/FreeFixed procedures.

Password. The file Password.Bcpl contains the Alto password routines, and can be used to do password checking in subsystems.

Keyboard. The keyboard handler is available in KeyStreamsB.Bcpl, KeyStreamsA.Asm and LevBuffer.asm. The procedure CreateKeyboardStream initializes the package, and returns a value (keys) that can be used as a keyboard stream.

Display. The display handler is available in the file DspStreamsB.Bcpl and DspStreamsA.Asm. Documentation is found later in this manual.

Directory. The file Dirs.Bcpl contains the directory manipulations described in section 3.5.

Fast Streams. The files FastStreamsB.bcpl and FastStreamsA.asm implement fast streams to memory. Documentation is part of DiskStreams.

Disk Streams. The files DiskStreams.bcpl, DiskStreamsMain.bcpl, and DiskStreamsAux.bcpl contain procedures for implementing disk streams. The fast file scanning facilities require the additional file DiskStreamsScan.bcpl. Documentation is found later in this manual.

Alloc. The file Alloc.Bcpl implements the allocator. See documentation later in this manual.

Basic File System. The files BfsInit.bcpl, BfsBase.Bcpl, BfsWrite.Bcpl, BfsCreate, BfsClose.bcpl, BfsDDMgr.bcpl, BfsNewDisk.bcpl and BfsFindHole.bcpl implement the basic file system (documentation appears later in this manual). They are maintained separately from the OS (sources: <AltoSource>BFSSources.dm; BRs: <Alto>BFSBRs.dm). They require Calendar.Asm, Dvec.Bcpl, Calls.Asm, BcplTricks.asm and SysErr.bcpl in order to operate.

Disk Streams: A Byte-Oriented Disk Input/Output Package

The disk streams package provides facilities for doing efficient sequential input/output to and from Alto disk files. It also includes operations for doing random positioning with moderate efficiency, and for performing various housekeeping operations. An introduction to streams can be found in the Alto Operating System Manual.

As part of these facilities, a "fast stream" capability permits very fast sequential byte access to objects stored in memory. An extension to the disk streams package permits reading of a disk stream to be overlapped with computation, thereby enabling the reading of files at full disk speed under favorable conditions.

The source files for the disk streams package are kept with the Alto Operating System in OS.DM:

Streams.D: public declarations;
DiskStreams.decl: private declarations;
FastStreamsB.bcpl and FastStreamsA.asm: Memory streams;
DiskStreams.bcpl: create/destroy a stream;
DiskStreamsMain.Bcpl: the 'main line' code;
DiskStreamsAux.bcpl: auxiliary disk stream functions;
DiskStreamsScan.bcpl: fast file scanning;
DiskStreamsOEP.bcpl: overlay entry point declarations.

The DiskStreams code (not the FastStreams code) may be swapped. To this end, the functions are distributed among three moderate-sized modules and intermodule references are minimized.

Streams use the generic procedures of a "disk object" to do disk transfers. The stream routines default the choice of disk to "sysDisk," a disk object created by the Alto operating system to provide access to the standard disk drive. However, you are free to open streams to other disks.

1. Data structures

The file Streams.D contains the public declarations of the disk streams package. Most users will not be concerned with these structures (except occasionally with their size, so as to be able to allocate the right amount of space for one of them), because the streams package provides procedures to perform all the operations which are normally needed.

The ST structure is common to all streams in the Alto operating system. It includes the procedures which implement the generic stream operations for this particular stream: Closes, Gets, Puts, Resets, Putbacks, Errors, and Endofs. In addition, there is a type, which for disk streams is always stTypeDisk, and three parameter words whose interpretation depends on the stream. The parameter words are not used by disk streams.

Fast streams are a specialization of streams, designed to quickly get or put bytes or words until a count is exhausted, and then call on a fixup routine which supplies a new count. Usually the count specifies the number of items remaining in a buffer, and the fixup routine empties or refills the buffer, but no such assumptions are made by fast streams. This facility is described in a later section; it is used by disk streams, but is of no concern to a program which simply wants to use disk streams.

A file pointer contains all the information required to access an Alto disk file. Its structure is described in detail in the Disks documentation. For a normal user of streams, a file pointer is simply a small structure which must be supplied to the CreateDiskStream routine to specify the file to which the stream should be attached. File pointers are normally obtained from directories, but a user is free to store them wherever he wishes.

A file address FA is a pointer to a specific byte in a file. It includes the address of the byte, divided into a page number (the page size depends on the disk in use; normally pages contain 512 bytes) and a byte number. It also includes a disk address, which is a hint as to the physical location of the specified page. Stream routines which use file addresses check the hint; if it turns out to be correct, they proceed, and otherwise they start at the beginning of the file and search for the desired page.

A complete file address CFA contains both a file pointer and a file address; it is a pointer to a specific byte anywhere in the file system.

A file position (FPOS) is a double-precision number which addresses a byte in a file. The first word is the most-significant half.

2. Properties of disk streams

All the stream procedures take as their first parameter a structure called a disk stream. A disk stream provides access to a file stored on the Alto disk. Each stream is associated with exactly one file, although it is possible to have several streams in existence at once which are associated with the same file. The file is a permanent object, which will remain on the disk until explicitly deleted. The stream is an ephemeral object, which goes away when it is closed, or whenever the Alto's memory is erased.

A file consists of a leader page, a length L, and a sequence of L bytes of data; each byte contains 8 bits. A stream is always positioned to some byte of the file, and the normal stream operations proceed sequentially from the current position to later positions in the file. The first byte is numbered 0. When the stream is positioned at byte n, this will be the next byte transferred by a Gets or Puts. There are also operations which reposition the stream. When data is written into the stream, the file is lengthened if necessary to make room for it. The file is never shortened except by TruncateDiskStream (which may be called by Closes; see below).

A stream can transact business a word at a time or a byte at a time, depending on how it is created. In the former case, if the length of the file is odd, the last word delivered will have garbage in its right byte.

You can replace the generic stream procedures if you wish (Gets, Puts, Closes, Resets, Errors, Endofs, Stateofs). The one you are most likely to want to replace is the error procedure. It is initialized to SysErr.

3. Procedures

This section describes the calling sequences and behavior of all the user-callable procedures in the streams package. If a parameter is followed by an expression in brackets, this means that the parameter will be defaulted to that expression if you supply 0. If the last few parameters you are supplying are defaulted, you can just omit them. Empty brackets mean that the parameter may be omitted. The parameter s stands for the disk stream the procedure works on.

Warning: Because the stream procedures occasionally use the RetryCall function, a procedure address cannot be computed, but must be the value of a static (global) or local variable. Thus "a>>proc(stream, b)" is not permitted, but "let pr=a>>proc; pr(stream, b)" is fine.

3.1. Creating and destroying

CreateDiskStream(filePtr, type [ksTypeReadWrite], itemSize [wordItem], Cleanup [Noop], errRtn [SysErr], zone [sysZone], nil, disk [sysDisk]) returns diskStream. A new disk stream is created and returned. It is associated with the file specified by filePtr on the given "disk," and positioned at item 0. Its type may be one of (see Streams.D for definitions):

ksTypeReadOnly
ksTypeWriteOnly
ksTypeReadWrite

Its itemSize may be one of (see Streams.D for definitions):

charItem
wordItem

If you supply a cleanup routine, it will be called with the stream as parameter just before the stream is destroyed by a Close. If returnOnCheckError is true, the routine will return 0 if the file id of the leader page at the address specified in the file pointer is different from the file id in the file pointer. You would want this if you wanted to use the file pointer as a hint, perhaps to be backed up by a directory lookup if it fails. In fact, the standard directory routine OpenFile does exactly that. If you supply a zone, it will be used to allocate the space needed by the stream. This space comes in two parts: the stream itself, about 60 words long, and the buffer, one page long.

Resets(s): flushes any buffers associated with the stream to the disk, and positions the stream to 0.

Closes(s): closes the stream, flushing the buffer and updating various information in the leader page if necessary. The last things it does are to call the cleanup routine passed to CreateDiskStream, and then to free the space for the stream. If the stream is open for writing only and it is not positioned at date byte 0, the file length is truncated to the current position.

CleanupDiskStream(s): flushes any buffers associated with the stream to the disk.

3.2. Transferring Data

Gets(s): returns the next item (byte or word, depending on the item size), or causes an error if there are no more items in the stream.

Puts(s, item): writes the next item into the stream. It causes an error if there is no more disk space, or if the stream was created read-only.

ReadBlock(s, address, count) returns actualCount: reads count words from the stream into memory, starting at the specified memory address. It returns the number of words actually read, which may be less than count if there were not enough words in the file. It never causes an end-of-file error. It is possible to use ReadBlock on a byte stream, but only if the stream is currently positioned at an even byte; otherwise there will be an error.

WriteBlock(s, address, count): writes count words from memory into the stream, starting at the specified memory address. The comment in ReadBlock about byte streams applies here also.

3.3. Reading state

Endofs(s): returns true if and only if there are no more items in the stream.

LnPageSize(s) returns the log (base 2) of the number of words in a page of the file.

FileLength(s, filePos []) returns lengthL: positions the file to its last byte and returns the length in bytes in filePos (FPOS), and the length mod 2**16 as its value.

FilePos(s, filePos []) returns posL: returns the current byte position in filePos (FPOS), and the current position mod 2**16 as its value.

GetCurrentFa(s, fileAddress) stores the current position in the file address (FA), including the disk address of the current page as a hint which can be used by JumpToFa.

GetCompleteFa(s, completeFileAddress) stores both the file pointer and the current position in the complete file address (CFA). This is enough information to create a stream (passing the file pointer to CreateDiskStream) and then to return to the current position (passing the file address to JumpToFa).

KsBufferAddress(s) returns address: returns the address in memory of the buffer for the stream. This is useful if you want to move the buffer; you can do so, and then reset the address with KsSetBufferAddress.

KsGetDisk(s) returns a pointer to the DSK object that describes the disk on which this stream is open (see Disks documentation).

KsHintLastPageFa(s) returns a pointer to a hint for the end of the file opened by stream s.

ReadLeaderPage(s, ld) reads the 256 word leader page for the file on which s is open into the vector pointed to by ld. The stream is left positioned at data byte 0.

3.4. Setting state

TruncateDiskStream(s) truncates the stream at its current position. Afterwards, Endofs(s) will be true.

PositionPage(s, page, doExtend [true]) returns wantedToExtend: positions the stream to byte 0 of the specified page. If doExtend is true, it will extend the file with zeros if necessary in order to make it long enough to contain the specified page. If doExtend is false, it will not do this, but will return true if it was unable to position the stream as requested because the file wasn't long enough. NOTE: This routine interprets "page" in the units associated with the disk on which the stream is open. If you wish a device-independent positioning command, see SetFilePos.

PositionPtr(s, byteNo, doExtend [true]) returns wantedtoExtend: positions the stream to the specified byte of the current page. DoExtend is interpreted exactly as for PositionPage.

JumpToFa(s, fileAddress) positions the file to the specified address (FA). It tries to use the disk address hint in the address, but falls back to PositionPage if that fails.

SetFilePos(s, filePos): positions the file to the byte specified by the double-precision number in filePos (FPOS).

SetFilePos(s, filePosH, filePosL): positions the file to the byte specified by the filePosH*2**16+filePosL.

KsSetBufferAddress(s, address): sets the buffer address to the specified memory address. It is the caller's responsibility to be sure that the buffer has the proper contents, and that it was allocated from the proper zone, so that when it is freed using the zone which was used by CreateDiskStream the right thing will happen.

ReleaseKs(s) will release all the storage used by the stream s, without referencing the disk at all. This is a way of aborting a stream, often useful when recovering from an unrecoverable disk error.

WriteLeaderPage(s, ld) writes the 256-word vector pointed to by ld on the leader page of the file on which s is open. The stream is left postioned at data byte 0.

3.5. File Scanning

The disk stream procedures described above have the property that they perform disk operations synchronously. When one of these procedures requires a disk transfer to be performed, it initiates the transfer and waits for it to complete. While certain procedures (e.g., ReadBlock, WriteBlock, SetFilePos, etc.) are capable of transferring many consecutive pages in a single disk operation, most stream routines are limited to one page per disk revolution. This performance is an order of magnitude below the raw transfer rate of the disk.

The procedures in the DiskStreamsScan module permit reading (but not writing) of a file to proceed at up to full disk speed, if the amount of computation to be performed per page is not too great (about 2 milliseconds). To make use of this facility, you must provide a certain amount of extra buffer space to be managed by the disk streams package, and you must take care of sequencing through the data in each page yourself rather than obtaining it one item at a time using Gets.

The flow of control is basically as follows. You create a disk stream in the normal fashion. When you want to start scanning the file, you pass the stream to InitScanStream, along with one or more additional page-size buffers, and it returns a Scan Stream Descriptor (SSD). Now, every time you want to examine the next page of the file, you call GetScanStreamBuffer, which returns a pointer to a buffer containing the contents of that page. The contents of the buffer remain valid until the next call to GetScanStreamBuffer. When you have scanned as much of the file as you care to, you call FinishScanStream, which destroys the SSD and leaves the stream positioned at the beginning of the page most recently returned by GetScanStreamBuffer. You should not execute any normal stream operations between the calls to InitScanStream and FinishScanStream.

InitScanStream(s, bufTable, nBufs) returns SSD. Creates a Scan Stream Descriptor in preparation for scanning the file corresponding to the stream s. bufTable is an array of pointers to page-size buffers, and nBufs is the number of buffers (there must be at least one). That is, the buffers are located at bufTable!0, bufTable!1, ..., bufTable!(nBufs-1). The SSD is allocated from the zone from which s was allocated. InitScanStream does not actually initiate any disk activity.

GetScanStreamBuffer(ssd) returns a pointer to a buffer containing the next page of the file being scanned, or zero if end-of-file has been reached. This procedure waits if necessary for the transfer of the next page to complete, and before returning it initiates as many new disk transfers as it has buffers for. The first page returned by GetScanStreamBuffer is the one at which the stream was positioned at the time InitScanStream was called. The initial portion of the SSD is a public structure (defined in Streams.d) containing the disk address, page number, and number of characters in the page most recently returned by GetScanStreamBuffer; you may use this information for whatever purposes you wish (e.g., in building up a file map for subsequent efficient random access to the stream).

FinishScanStream(ssd) waits for disk activity to cease, updates the state in the corresponding stream, and destroys the SSD. The stream is left positioned at the beginning of the last page returned by GetScanStreamBuffer, or at end-of-file if GetScanStreamBuffer most recently returned zero.

The package uses the stream buffer in addition to the buffers passed explicitly to InitScanStream. It is possible to scan a file at full disk speed (assuming the file is consecutively allocated) with two buffers (i.e., just one additional buffer), so long as the interval between calls to GetScanStreamBuffer is no greater than 3.3 milliseconds (or about 2 milliseconds of computation on the caller's part). If more computation per page is required, or the amount of computation per page is highly variable, then more buffers are required to maintain maximum throughput.

4. Fast Streams

A fast stream structure must begin with the structure declared as FS in Streams.D; following this you can put anything you like. To initialize this structure, use

InitializeFstream(s, itemSize, PutOverflowRoutine, GetOverflowRoutine, GetControlCharRoutine [Noop]). The s paramter points to storage for the stream structure, lFS words long. The itemSize is as for CreateDiskStream. The overflow routines are explained below. GetControlCharRoutine(item, s) will be called whenever a Gets for a charItem stream is about to return an item between 0 and #37, and its value is returned as the value of the Gets. The initialization provides Gets, Puts, and Endofs routines; the other stream procedures are left as Errors.

SetupFstream(s, wordBase, currentPos, endPos) is used to set up a fast stream to transfer data to or from a buffer in memory. WordBase is the address of the buffer in memory, and currentPos and endPos are byte addresses in the buffer. CurrentPos is the address of the first byte to be transferred, and endPos is the address of the first byte which should not be transferred. CurrentPos is rounded up to a word if the item size is wordItem, and endPos is rounded up to a word.

When a Gets or Puts attempts to transfer the byte addressed by endPos, the corresponding overflow routine is called, with the same parameters that were passed to the Gets or Puts. The overflow routine can do one of two things:

do the work and return

fix things up so that the Gets or Puts can succeed, and then exit with RetryCall(stream, item).

SetEof(s, newValue) sets the end-of-file flag in the stream. When this flag is set, the Gets routine is replaced by a routine which gives an end-of-file error, and when it is cleared, the old Gets routine is restored.

CurrentPos(s) returns the current position in the buffer, always measured in bytes.

ItemSize(s) returns the item size of the stream.

Dirty(s) returns true if the dirty flag is true. This flag is set to true whenever a Puts is done.

SetDirty(s, value) sets the dirty flag to the specified value (true or false).

5. Errors

Whenever an operation on a stream causes an error, the error procedure in the stream is called with two parameters: the stream, and an error code. The error procedure is initialized to SysErr, but you can change it to whatever you like. The error codes for errors generated by the disk stream package are:

- 1301 illegal item size to CreateDiskStream or InitializeFstream
- 1302 end of file
- 1303 attempt to execute an undefined stream operation
- 1200 attempt to write a read-only stream
- 1201 attempt to do ReadBlock or WriteBlock on a stream not positioned at a word.
- 1202 attempt to PositionPointer outside the range [0 .. #1000]
- 1203 attempt to do a disk operation on something not a disk stream
- 1204 bug in disk streams package
- 1205 CreateDiskStream cannot allocate space for the stream from the zone supplied

Display stream package

A library package is now available which provides display streams of great flexibility. Special features include multiple fonts, repositioning to any bit position in the current line (or, under proper circumstances, any line), selective erasing and polarity inversion, and better utilization of the available bitmap space.

The package consists of two files, DspStreamB.Bcpl and DspStreamA.Asm. In addition, files Streams.d and AltoDefs.d provide useful parameter and structure declarations, in particular the parameters lDCB and lDS mentioned below. The package does not require any routines other than those in the operating system.

1. Creating a display stream

CreateDisplayStream(nLines, pBlock, lBlock, Font [sysFont], wWidth [38], Options [DScompactleft+DScompactright], zone [sysZone]): creates a display stream. nLines is the maximum number of lines that will be displayed at once: it is completely independent of the amount of space supplied for bitmap and DCBs. pBlock is the beginning address of storage that can be used for the display bitmap and control blocks; its length is lBlock. This block may be shortened slightly in order to align things on even word boundaries. Font is a pointer to the third word of a font in AL format to use for the stream. wWidth gives the width of the screen in Alto screen units, divided by 16; it must be an even number. Zone is a free-space pool from which any additional space needed by the stream can be seized. (For a description of zones, see the Alto OS manual.)

The minimum space for a display stream is lDCB*nLines+fh*wWidth+1, where fh is the height of the standard system font, rounded up to an even number; the +1 allows the display stream package to align the space on an even word boundary. This, however, only provides enough bitmap for a single line. A space allocation of lDCB*nLines+fh*wWidth*nLines+1 guarantees enough bitmap for all nLines lines. The display stream package uses all the available space and then, if necessary, blanks lines starting from the top to make room for new data.

Options, if supplied, controls the action of the stream under various exceptional conditions. The various options have mnemonic names (defined in Streams.d) and may be added together. Here is the list of options:

DScompactleft	allows the bitmap space required for a line to be reduced when scrolling by eliminating multiples of 16 initial blank bit positions and replacing them with the display controller's "tab" feature. However, a line in which this has occurred may not be overwritten later (with SetLinePos, see below).
DScompactright	allows the bitmap space for a line to be reduced when scrolling by eliminating multiples of 16 blank bit positions on the right. Overwriting is allowed up to the beginning of the blank space, i.e. you cannot make a line longer by overwriting if you select this option.
DSstopright	causes characters to be discarded when a line becomes full, rather than scrolling onto a new line.
DSstopbottom	causes characters to be discarded in preference to losing data from the screen. This applies when either all nLines lines are occupied, or when the allocated bitmap space becomes full.
DSnone	none of the above (this option is necessary so that 0 defaults to DScompactleft+DScompactright).

2. Displaying the stream contents

ShowDisplayStream(s, how [DSbelow], otherStream [dsp]): This procedure controls the presentation of a chain of display control blocks on the screen. If how id DSbelow, the stream will be displayed immediately below otherStream; if DSabove, immediately above; if DSalone, it will be the only stream displayed; if DSdelete, the stream s will be removed from the screen. The third argument is not needed for DSalone or DSdelete.

If you wish to construct your own "stream" for purposes of passing it to ShowDisplayStream, it is sufficient that s>>DS.fdcb point to the first DCB of a list and that s>>DS.ldcb point to the last DCB. These are the only entries referenced by ShowDisplayStream (note that fdcb and ldcb are the first two words of a stream structure).

3. Current-line operations

ResetLine(ds): erases the current line and resets the current position to the left margin.

GetFont(ds): returns the current font of ds.

SetFont(ds, pfont): changes the font of the display stream ds. Pfont is a pointer to word 2 of a font, which is compatible with GetFont. Characters which have been written into the stream already are not affected; future characters will be written in the new font. If the font is higher than the font initially specified, writing characters may cause unexpected alteration of lines other than the line being written into. if pFont!-2 is negative, then pFont!-1 is a pointer to a font (word 3, remember) and subsequent characters put to the stream will be shown in synthetic bold face by scan converting the character, moving over one bit and scan converting it again.

GetBitPos(ds): returns the bit position in the current line. The bit position is normally initialized to 8.

SetBitPos(ds, pos): sets the bit position in the current line to pos and returns true, if pos is not too large; otherwise, returns false. Pos must be less than 606 (the display width) minus the width of the widest character in the current font. Resetting the bit position does not affect the bitmap; characters displayed at overlapping positions will be "or"ed in the obvious manner.

EraseBits(ds, nbits, flag): changes bits in ds starting from the current position. Flag=0, or flag omitted, means set bits to 0 (same as background); flag=1 means set bits to 1 (opposite from background); flag=-1 means invert bits from their current state. If nbits is positive, the affected bits are those in positions pos through pos+nbits-1, where pos is GetBitPos(ds); if nbits is negative, the affected positions are pos+nbits through pos-1. In either case, the final position of the stream is pos+nbits.

Here are two examples of the use of EraseBits. If the last character written on ds was ch, EraseBits(ds, -CharWidth(ds, ch)) will erase it and back up the current position (see below for CharWidth). If a word of width ww has just been written on ds, EraseBits(ds, -ww, -1) will change it to white-on-black.

4. Inter-line operations

GetLinePos(ds): returns the line number of the current line; the top line is numbered 0. Unlike the present operating system display streams, which always write into the bottom line and scroll up, the display streams provided by this package start with the top line and only scroll when they reach the bottom.

SetLinePos(ds, pos): sets the current line position in ds to pos. If the line has not yet been written into, or if it has zero width, or if it is indented as the result of compacting on the left, SetLinePos has no effect and returns false; otherwise, SetLinePos returns true. Note that if you want to get back to where you were before, you must remember where that was (using GetLinePos and GetBitPos).

InvertLine(ds, pos): Inverts the black/white sense of the line given by pos. Returns the old sense (0 is black-on-white).

ds>>DS.cdcb: points to the DCB for the current line. You may (at your own risk) fiddle with this to achieve various effects.

5. Scrolling

The display stream package writes characters using a very fast assembly language routine until either the current line is full or it encounters a control character. In either of these situations it calls a scrolling procedure whose address is a component of the stream. The scrolling procedure is called with the same arguments as PUTS, i.e. (ds, char), and is expected to do whatever is necessary. The standard procedure takes the following action:

1) Null (code 0) is ignored.
2) New line (code 15b) causes scrolling.
3) Tab (code 11b) advances the bit position to the next multiple of 8 times the width of "blank" (code 40b) in the current font: if this would exceed the right margin, just puts out a blank.
4) Other control characters (codes 1-10b, 12b-14b, 16b-37b) print with whatever symbol appears in the font.
5) If a character will not fit on the current line, scrolling occurs and the character is printed at the beginning of the new line (unless the DSstopright option was chosen, in which case the character is simply discarded).

The scrolling procedure is also called with arguments (ds, -1) whenever a contemplated scrolling operation would cause information to disappear from the screen, either because nLines lines are already present or because the bitmap space is full (unless the DSstopbottom option was chosen, in which case the procedure is not called and the action is the same as if it had returned false). If the procedure returns true, the scrolling operation proceeds normally. If the procedure returns false, the scrolling does not take place, and the character which triggered the operation is discarded.

The user may supply a different scrolling procedure simply by filling it into the field ds>>DS.scroll.

6. Miscellaneous

GetLmarg(ds): returns the left margin position of ds. The left margin is initialized to 8 (about 1/10" from the left edge of the screen).

SetLmarg(ds, pos): sets the left margin of ds to pos.

GetRmarg(ds): returns the right margin position of ds. The right margin is initialized to the right edge of the screen: this is the value of the displaywidth parameter in DISP.D.

SetRmarg(ds, pos): sets the right margin of ds to pos.

CharWidth(StreamOrFont, char): returns the width of the character char in the stream StreamOrFont; if StreamOrFont is not a stream, it is assumed to be a font pointer.

Alloc -- A Basic Storage Allocator

The Alloc package contains a small and efficient non-relocating storage allocator. It doesn't do much, but what it does it does very well. Initially the user gives the allocator one (or several) blocks of storage by calls on InitializeZone. The user can later add storage to a zone by calling AddToZone. The function Allocate returns a pointer to a block allocated from a given zone. Calling Free returns a previously-allocated block to a given zone.

Argument lists given below are decorated with default settings. An argument followed by [exp] will default if omitted or zero to the value exp; an argument followed by [...exp] will default if omitted to exp.

InitializeZone, AddToZone

The function zone = InitializeZone(Zone, Length, OutOfSpaceRoutine [...SysErr], MalFormedRoutine [...SysErr]) initializes the block of storage beginning at address Zone and containing Length words to be a free storage zone. OutOfSpaceRoutine is taken to be an error handling routine that will be called whenever a requested allocation cannot be satisfied. MalFormedRoutine is an error printing routine that is called whenever the Alloc package detects an error in the consistency of the zone data structure. InitializeZone builds the zone data structure, and returns a pointer to a "zone," which is used for all subsequent calls to Allocate and Free for the zone.

The function AddToZone(Zone, Block, Length) adds the block of storage beginning at Block and containing Length words to the zone pointed to by Zone.

Alloc restricts the maximum size of the blocks it will allocate and of the "Length" arguments for InitializeZone and AddToZone to 32K-1.

Allocate, Free

The function Allocate(Zone, Length, returnOnNoSpace [...0], Even [...0]) allocates a block of Length words from Zone and returns a pointer to that block. If the allocation cannot be done, one of two cases pertains: (1) returnOnNoSpace is non-zero or the OutOfSpaceRoutine provided for the zone is 0: Allocate returns the value 0; if returnOnNoSpace is not -1, the size of the largest available block is stored in @returnOnNoSpace; (2) otherwise, the value returned to the caller is the result of OutOfSpaceRoutine(Zone, ecOutOfSpace, Length).

If the optional parameter Even is true, the block allocated will be guaranteed to begin on an even word boundary. This is useful when allocating display buffers.

The procedure Free(Zone, Block) gives a previously-allocated block of storage back to the zone pointed to by Zone. Block must have been the value of a call on Allocate.

CheckZone

The Alloc package contains considerable facilities for debugging. Conditional compilation will enable various levels of consistency checking; the remainder of this paragraph assumes that the checking is enabled. Users should consult the source file (Alloc.Bcpl) for details concerning the conditional compilation.

The procedure CheckZone(zone), which may be called conveniently from Swat, will perform a fairly exhaustive consistency check of the zone (provided that conditional compilation has caused the code to be present!).

In addition, certain checking will be performed on the various calls to the package, provided that the MalFormedRoutine parameter supplied for the zone is non-zero.

If an error is detected, the call MalFormedRoutine(zone, errCode) is executed. Values of the error code are:

ecOutOfSpace	1801	Not enough space to satisfy a request.
ecZoneAdditionError	1802	Too large or too small addition to zone.
ecBlockNotAllocated	1803	Free has been called with a bad block.
ecIllFormed	1804	The consistency-checker has found some error in the zone. Consult Alloc.Bcpl.

Free-Standing Zones

It is often desirable to use a single 16-bit quantity to describe an entire free-space pool, together with its allocating and freeing procedures. For example, one can pass to the operating system such a quantity; the system can thereafter acquire and release space without knowing the details of how the operations are done. The zones constructed by Alloc have this property:

```
zone>>ZN.Allocate(zone, Length) will allocate a block
zone>>ZN.Free(zone, Block)      will free a block
```

By convention, these entries are at the beginning of a zone. Thus, all you need to know about the ZN data structure is:

```
structure ZN[
    Allocate    word  //Allocation procedure
    Free        word  //Free procedure
    ...rest of zone...
    ]
```

Example

The following terrible implementation of the factorial function illustrates the use of Alloc:

```
static [ Spare
    SpareIsAvail
    FactZone
    ]

let Factorial(n) = valof
    [ let FactZoneV = vec 256
    let MySpare = vec 37
    Spare = MySpare
    SpareIsAvail = true

    FactZone = InitializeZone(FactZoneV, 256, StkOvfl)

    let FactVal = InnerFact(n)

    resultis FactVal
    ]

and InnerFact(n) = valof
    [ structure STKENT:
        [ link word
        value word
        ]

    manifest [ empty = -1;
        wordsize = 16
        ]
```

```
        let stack = empty

        while n gr 1 do
            [ let stkent = Allocate(FactZone, size STKENT/wordsize)
            stkent>>STKENT.link = stack
            stkent>>STKENT.value = n
            stack = stkent
            n = n-1
            ]

        let value = 1

        while stack ne empty do
            [ value = value*(stack>>STKENT.value)
            let stkent = stack
            stack = stkent>>STKENT.link
            Free(FactZone, stkent)
            ]

        resultis value
        ]

    and StkOvfl(Zone, nil, Length) = valof
        [ unless SpareIsAvail do
            [ Ws("Aargh! Stack stuck!")
            finish
            ]
        AddToZone(FactZone, Spare, 37)
        SpareIsAvail = false
        resultis Allocate(FactZone, Length)
        ]
```

Disks: The Alto File System

This document describes the disk formats used in the Alto File System. It also describes a "disk object," a Bcpl software construct that is used to interface low-level disk drivers with packages that implement higher-level objects, such as streams.

The primary focus of the description will be for the "standard" Alto disks: either (1) up to 2 Diablo Model 31 disk drives or (2) one Diablo Model 44 disk drive. The low-level drivers for these disks are called "Bfs" (Basic File System). With minor modifications, the description below applies to the Trident Model T80 and T300 disk drives, when formatted for Alto file system conventions. The differences are flagged with the string [Trident]. Low-level drivers for the Trident disks are called "Tfs."

1. Distribution

Relocatable binary files for the BFS are kept in <Alto>BFSBrs.dm. The sources, command files, and test program (described later in this document) are kept in <AltoSource>BFSSources.dm Relocatable binary files for the TFS are kept in <Alto>TFS.dm; sources are kept on <AltoSource>TFSSources.dm.

2. File and Disk Structure

This section describes the conventions of the Alto file system. The files AltoFileSys.D and Bfs.D contain Bcpl structure declarations that correspond to this description ([Trident]: See also "Tfs.D").

The unit of transfer between disk and memory, and hence that of the file system, is the disk sector. Each sector has three fields: a 2-word header, an 8-word label, and a 256-word data page. ([Trident]: The fields are a 2-word header, a 10-word label, and a 1024-word data page.)

A sector is identified by a disk address; there are two kinds of disk addresses, real and virtual. The hardware deals in real addresses, which have a somewhat arbitrary format. An unfortunate consequence is that the real addresses for all the pages on a disk unit are sparse in the set of 16 bit integers. To correct this defect, virtual addresses have been introduced. They have the property that the pages of a disk unit which holds n pages have virtual addresses 0 ... (n-1). Furthermore, the ordering of pages by virtual address is such that successive pages in the virtual space are usually sequential on the disk. As a result, assigning a sequence of pages to consecutive virtual addresses will ensure that they can be read in as fast as possible.

2.1. Legal Alto Files

An Alto file is a data structure that contains two sorts of information: some is mandatory, and is required for all legal files; the remainder is "hints". Programs that operate on files should endeavor to keep the hints accurate, but should never depend on the accuracy of a hint.

A legal Alto file consists of a sequence of pages held together by a doubly-linked list recorded in the label fields. Each label contains the mandatory information:

The forward and backward links, recorded as real disk addresses.

A page number which gives the position of the page in the file; pages are numbered from 0.

A count of the number of characters of data in the page (numchars). This may range from 0 (for a completely empty page) to 512 (for a completely full page). ([Trident]: A full page contains 2048 characters.)

A real file id, which is a three-word unique identifier for the file. The user normally deals with virtual file ids (see the discussion of file pointers, below), which are automatically converted into real file ids when a label is needed.

Three bits in the file id deserve special mention:

Directory: This bit is on if the file is itself a directory file. This information is used by the disk Scavenger when trying to re-build a damaged disk data structure.

Random: This bit is currently unused.

NoLog: This bit is no longer used, but many existing files are likely to have it set.

Leader Page: Page 0 of a file is called the leader page; it contains no file data, but only a collection of file properties, all of which are hints. The structure LD in AltoFileSys.D declares the format of a leader page, which contains the following standard items:

The file name, a hint so that the Scavenger can enter this file in a directory if it is not already in one.

The times for creation, last read and last write, interpreted as follows:

A file's creation date is a stamp generated when the information in the file is created. When a file is copied (without modification), the creation date should be copied with it. When a file is modified in any way (either in-place or as a result of being overwritten by newly-created information), a new creation date should be generated.

A file's write date is updated whenever that file is physically written on a given file system.

A file's read date is updated whenever that file is physically read from within a given file system.

A pointer to the directory in which the file is thought to be entered (zeroes imply the system directory SysDir).

A "hint" describing the last page of the file.

A "consecutive" bit which is a hint that the pages of the file lie at consecutive virtual disk addresses.

The changeSerial field related to version numbering: whenever a new version of a file "foo" is made, the changeSerial field of all other files "foo" (i.e., older versions) is incremented. Thus, a program that wishes to be sure that it is using the most recent version of a file can verify that changeSerial=0. If a program keeps an FP as a hint for a file, and is concerned about the relative position of that file in the list of version numbers, it can also keep and verify the changeSerial entry of the file. Version numbers have been deimplemented.

These standard items use up about 40 words of the leader page. The remaining space is available for storing other information in blocks which start with a one word header containing type and length fields. A zero terminates the list. The structure FPROP in AltoFileSys.d defines the header format. The only standard use of this facility is to record the logical shape of the disk in the leader page of SysDir.

Data: The first data byte of a file is the first byte of page 1.

In a legal file with n pages, the label field of page i must contain:

A next link with the real disk address of page (i+1), or 0 if i=n-1.

A previous link with the real disk address of page (i-1), or 0 if i=0.

A page number between 0 and (n-1), inclusive.

A numchars word = 512 if i<n-1, and <512 if i=n-1. The last page must not be completely full. ([Trident]: = 2048 if i<n-1, and <2048 if i = n-1.)

A real file id which is the same for every page in the file, and different from the real file id of any other file on the disk.

A file is addressed by an object called a file pointer (FP), which is declared in AltoFileSys.D. A file pointer contains a virtual file id, and also the virtual address of the leader page of the file. The low-level disk routines construct a real file id from the virtual one when they must deal with a disk label. Since it is possible for the user to read a label from the disk and examine its contents, the drivers also provides a routine which will convert the real file id in the label into a file pointer (of course, the leader address will not be filled in).

Note: Real disk address 0 (equal virtual disk address 0) cannot be part of any legal Alto file because the value 0 is reserved to terminate the forward and backward chains in sector labels. However, disk address 0 is used for "booting" the Alto: when the boot key is pressed when no keyboard keys are down, sector 0 is read in as a bootstrap loader. The normal way to make a file the "boot file" is to first create a legal Alto file with the bootstrap loader as the first data page (page 1), and then to copy this page (label and data) into disk sector 0. Thus the label in sector 0 points forward to the remainder of the boot file.

2.2. Legal Alto Disks

A legal disk is one on which every page is either part of a legal file, or free, or "permanently bad." A free page has a file id of all ones, and the rest of its label is indeterminate. A permanently bad page has a file id with each of the three words set to -2, and the remainder of the label indeterminate.

2.3. Alto Directory Files

A directory is a file for associating string names and FP's. It has the directory bit set in its file id, and has the following format (structure DV declared in AltoFileSys.D).

It is a sequence of entries. An entry contains a header and a body. The length field of the header tells how many words there are in the entry, including the header. The interpretation of the body depends on the type, recorded in the header.

dvTypeFree=0: free entry. The body is uninterpreted.

dvTypeFile=1: file entry. The body consists of a file pointer, followed by a Bcpl string containing the name of the file. The file name must contain only upper and lower case letters, digits, and characters in the string "+-.!$". They must terminate with a period (".") and not be longer than maxLengthFn characters. If there are an odd number of bytes in the name, the "garbage byte" must be 0. The interpretation of exclamation mark (!) is special; if a file name ends with ! followed only by digits (and the mandatory "."), the digits specify a file version number.

The main directory is a file with its leader page stored in the disk page with virtual address 1. There is an entry for the main directory in the main directory, with the name SysDir. All other directories can be reached by starting at the main directory.

2.4. Disk Descriptor

There is a file called DiskDescriptor entered in the main directory which contains a disk descriptor structure which describes the disk and tells which pages are free. The disk descriptor has two parts: a 16 word header which describes the shape of the disk, and a bit table indexed by virtual disk address. The declaration of the header structure is in AltoFileSys.D.

The "defaultVersionsKept" entry in the DiskDescriptor records the number of old versions of files that should be retained by the system. If this entry is 0, no version accounting is done: new files simply replace old ones. Version numbers have been deimplemented.

The entry in the disk descriptor named "freePages" is used to maintain a count of free pages on the disk. This is a hint about a hint: it is computed when a disk is opened by counting the bits in the bit table, and then incrementing and decrementing as pages are released and allocated. However the bit table is itself just a collection of hints, as explained below.

The bit table contains a "1" corresponding to each virtual disk address that is believed to be occupied by a file, and "0" for free addresses. These values are, however, only hints. Programs that assign new pages should check to be sure that a page thought to be free is indeed so by reading the label and checking to see that it describes a free page. (The WriteDiskPages and CreateDiskFile procedures in the disk object perform this checking for you.)

2.5. Oversights

If the Alto file system were to be designed again, several deficiencies could be corrected:

> Directory entries and label entries should have the same concept of file identifier. Presently, we have filePointers and fileIds.
>
> There is no reason why the last page of a file cannot contain 512 bytes.
>
> It is unfortunate that the disk controller will not check an entry of 0 in a label, because these values often arise (numChars of the last page, page number of the leader page). Another don't care value should be chosen: not a legal disk address; with enough high order bits so that it will check numChars and page number fields.
>
> The value used to terminate the chain of disk addresses stored in the labels should not be a legal disk address. (It should also not be zero, so that it may be checked.) If it is a legal address, and if you try to run the disk at full speed using the trick of pointing page i's label at page i+1's disk address in the command block, the disk will try to read the page at the legal disk address represented by the chain terminator. Only when this results in an error is end of file detected. A terminator of zero has the undesirable property that a seek to track 0 occurs whenever a chain runs into end-of-file.

3. The Disk Object

In order to facilitate the interface between various low-level disk drivers and higher-level software, we define a "disk object." A small data structure defines a number of generic operations on a disk -- the structure DSK is defined in "Disks.D." Each procedure takes the disk structure as its first argument:

> ActOnDiskPages: Used to read and write the data fields of pages of an existing file.
>
> WriteDiskPages: Used to read and write data fields of the pages of a file, and to extend the file if needed.
>
> DeleteDiskPages: Used to delete pages from the end of a file.
>
> CreateDiskFile: Used to create a new disk file, and to build the leader page correctly.
>
> AssignDiskPage: Used to find a free disk page and return its virtual disk address.
>
> ReleaseDiskPage: Used to release a virtual disk address no longer needed.
>
> VirtualDiskDA: Converts a real disk address into a virtual disk address.

RealDiskDA: Converts a virtual disk address into a real disk address.

InitializeDiskCBZ: Initializes a Command Buffer Zone (CBZ) for managing disk transfers.

DoDiskCommand: Queues a Command Buffer (CB) to initiate a one-page transfer.

GetDiskCb: Obtains another CB, possibly waiting for an earlier transfer to complete.

CloseDisk: Destroys the disk object.

In addition, there are several standard data entries in the DSK object:

fpSysDir: Pointer to the FP for the directory on the disk. (This always has a constant format -- see discussion above.)

fpDiskDescriptor: Pointer to the FP for the file "DiskDescriptor" on the disk.

fpWorkingDir: Pointer to the FP to use as the "working directory" on this disk. This is usually the same as fpSysDir.

nameWorkingDir: Pointer to a Bcpl string that contains the name of the working directory.

lnPageSize: This is the log (base 2) of the number of words in a data page on this disk.

driveNumber: This entry identifies the drive number that this DSK structure describes.

retryCount: This value gives the number of times the disk routines should retry an operation before declaring it an error.

totalErrors: This value gives a cumulative count of the number of disk errors encountered.

diskKd: This entry points to a copy of the DiskDescriptor in memory. Because the bit table can get quite large, only the header needs to be in memory. This header can be used, for example, to compute the capacity of the disk.

lengthCBZ, lengthCB: The fixed overhead for a CBZ and the number of additional words required per CB.

In addition to this standard information, a particular implementation of a disk class may include other information in the structure.

4. Data Structures

The following data structures are part of the interface between the user and the disk class routines:

pageNumber: as defined in the previous section. The page number is represented by an integer.

DAs: a vector indexed by page number in which the ith entry contains the virtual disk address of page i of the file, or one of two special values (which are declared as manifest constants in Disks.D):

eofDA: this page is beyond the current end of the file;
fillInDA: the address of this page is not known.

Note that a particular call on the file system will only reference certain elements of this vector, and the others do not have to exist. Thus, reading page i will cause references only to DAs!i and DAs!(i+1), so the user can have a two-word vector v to hold these quantities, and pass v-i to the file system as DAs.

CAs: a vector indexed by page number in which the ith entry contains the core address to or from which page i should be transfered. The note for DAs applies here also.

fp (or filePtr): file pointer, described above. In most cases, the leader page address is not used.

action: a magic number which specifies what the disk should do. Possible values are declared as manifest constants in Disks.D:

DCreadD:	check the header and label, read the data;
DCreadLD:	check the header, read the label and data;
DCreadHLD:	read the header, label, and data;
DCwriteD:	check the header and label, write the data;
DCwriteLD:	check the header, write the label and data;
DCwriteHLD:	write the header, label, and data;
DCseekOnly:	just seek to the specified track
DCdoNothing:	

A particular implementation of the disk class may also make other operations available by defining additional magic numbers.

5. Higher-level Subroutines

There are two high-level calls on the basic file system:

pageNumber = ActOnDiskPages(disk, CAs, DAs, filePtr, firstPage, lastPage, action, lvNumChars, lastAction, fixedCA, cleanupRoutine, lvErrorRoutine, returnOnCheckError, hintLastPage).

Parameters beyond "action" are optional and may be defaulted by omitting them or making them 0.

Here firstPage and lastPage are the page numbers of the first and last pages to be acted on (i.e. read or written, in normal use). This routine does the specified action on each page and returns the page number of the last page successfully acted on. This may be less than lastPage if the file turns out to have fewer pages. DAs!firstPage must contain a disk address, but any of DAs!(firstPage+1) through DAs!(lastPage+1) may be fillInDA, in which case it will be replaced with the actual disk address, as determined from the chain when the labels are read. Note that the routine will fill in DAs!(lastPage+1), so this word must exist.

The value of the numChars field in the label of the last page acted on will be left in rv lvNumChars. If lastAction is supplied, it will be used as the action for lastPage instead of action. If CAs eq 0, fixedCA is used as the core address for all the data transfers. If cleanupRoutine is supplied, it is called after the successful completion of each disk command, as described below under "Lower-level disk access". (Note: providing a cleanup routine defeats the automatic filling in of disk addresses in DAs).

Disk transfers that generate errors are retried several times and then the error routine is called with
rv lvErrorRoutine(lvErrorRoutine, cb, errorCode)

In other words, lvErrorRoutine is the address of a word which contains the (address of the) routine to be called when there is an error. The errorCode tells what kind of error it was; the standard error codes are tabulated in a later section. The cb is the control block which caused the error; its format depends on the particular implementation of the drivers (Bfs: the structure CB in Bfs.D).

The intended use of lvErrorRoutine is this. A disk stream contains a cell A, in a known place in the stream structure, which contains the address of a routine which fields disk errors. The address of A is passed as lvErrorRoutine. When the error routine is called, it gets the address of A as a parameter, and by subtracting the known position of A in the disk stream structure, it can obtain the address of the stream structure, and thus determine which stream caused the error.

The default value of returnOnCheckError is false. If returnOnCheckError is true and an error is encountered, ActOnDiskPages will not retry a check error and then report an error. Instead, it will return -(#100+i), where i is the page number of the last page successfully transferred. This feature allows ActOnDiskPages to be used when the user it not sure whether the disk address he has is correct. It is used by the disk stream and directory routines which take hints; they try to read from the page addressed by the hint with returnOnCheckError true, and if they get a normal return they know that the hint was good. On the other hand, if it was not good, they will get the abnormal return just described, and can proceed to try again in a more conservative way.

The hintLastPage argument, if supplied, indicates the page number of what the caller believes to be the last page of the file (presumably obtained from the hint in the leader page). If the hint is correct, ActOnDiskPages will ensure that the disk controller does not chain past the end of the file and seek to cylinder zero (as described earlier under "Oversights"). If the hint is incorrect, the operation will still be performed correctly, but perhaps with a loss in performance. Note that the label is not rewritten by DCwriteD, so that the number of characters per page will not change. If you need to change the label, you should use WriteDiskPages unless you know what you are doing.

ActOnDiskPages can be used to both read and write a file as long as the length of the file does not have to change. If it does, you must use WriteDiskPages.

pageNumber = WriteDiskPages(disk, CAs, DAs, filePtr, firstPage, lastPage, lastAction, lvNumChars,
lastNumChars, fixedCA, nil, lvErrorRoutine, nil, hintLastPage).

Arguments beyond lastPage are optional and may be defaulted by omitting them or making them 0 (but lastNumChars is not defaulted if it is 0).

This routine writes the specified pages from CAs (or from fixedCA if CAs is 0, as for ActOnDiskPages). It fills in DAs entries in the same way as ActOnDiskPages, and also allocates enough new pages to complete the specified write. The numChars field in the label of the last page will be set to lastNumChars (which defaults to 512 [Trident]: 2048). It is generally necessary that DAs!firstPage contain a disk address. The only situation in which it is permissible for DAs!firstPage to contain fillInDA is when firstPage is zero and no pages of the file yet exist on the disk (i.e., when creating page zero of a new file).

In most cases, DAs!(firstPage-1) should have the value which you want written into the backward chain pointer for firstPage, since this value is needed whenever the label for firstPage needs to be rewritten. The only case in which it doesn't need to be rewritten is when the page is already allocated, the next page is not being allocated, and the numChars field is not changing.

If lastPage already exists:

1) the old value of the numChars field of its label is left in rv lvNumChars.

2) if lastAction is supplied, it is applied to lastPage instead of DCwriteD. It defaults to DCwriteD.

WriteDiskPages handles one special case to help in "renaming" files, i.e. in changing the FP (usually the serial number) of all the pages of a file. To do this, use ActOnDiskPages to read a number of pages of the file into memory and to build a DAs array of valid disk addresses. Then a call to WriteDiskPages with lastAction=-1 will write labels and data for pages firstPage through lastPage (DAs!(firstPage-1) and DAs!(lastPage+1) are of course used in this writing process). The numChars field of the label on the last page is set to lastNumChars. To use this facility, the entire DAs array must be valid, i.e. no entries may be fillInDA.

In addition to these two routines, there are two others which provide more specialized services:

CreateDiskFile(disk, name, filePtr, dirFilePtr, word1 [0], useOldFp [false], pageBuf[0])

Creates a new disk file and writes its leader page. It returns the serial number and leader disk address in the FP structure filePtr. A newly created file has one data page (page 1) with numChars eq 0.

The arguments beyond filePtr are optional, and have the following significance:

If dirFilePtr is supplied, it should be a file pointer to the directory which owns the file. This file pointer is written into the leader page, and is used by the disk Scavenger to put the file back into the directory if it becomes lost. It defaults to the root directory, SysDir.

The value of word1 is "or"ed into the filePtr>>FP.serialNumber.word1 portion of the file pointer. This allows the directory and random bits to be set in the file id.

If useOldFp is true, then filePtr already points to a legal file; the purpose of calling CreateDiskFile is to re-write all the labels of the existing file with the new serial number, and to re-initialize the leader page. The data contents of the original file are lost. Note that this process effectively "deletes" the file described by filePtr when CreateDiskFile is called, and makes a new file; the FP for the new file is returned in filePtr.

If pageBuf is supplied, it is written on the leader page of the new file after setting the creation date and directory FP hint (if supplied). If pageBuf is omitted, a minimal leader page is created.

DeleteDiskPages(disk, CA, firstDA, filePtr, firstPage, newFp, hintLastPage)

Arguments beyond firstPage are optional. Deletes the pages of a file, starting with the page whose number is firstPage and whose disk address is firstDA. CA is a page-sized buffer which is clobbered by the routine. hintLastPage is as described under ActOnDiskPages.

If newFp is supplied and nonzero, it (rather than freePageFp) is installed as the FP of the file, and the pages are not deallocated.

6. Allocating Disk Space

The disk class also contains routines for allocating space and for converting between virtual and real disk addresses. In most cases, users need not call these routines directly, as the four routines given above (ActOnDiskPages, WriteDiskPages, DeleteDiskPages, CreateDiskFile) manage disk addresses and disk space internally.

AssignDiskPage(disk, virtualDA, nil) returns the virtual disk address of the first free page following virtualDA, according to the bit table, and sets the corresponding bit. It does not do any checking that the page is actually free (but WriteDiskPages does). If there are no free pages it returns -1. If it is called with three arguments, it returns true if (virtualDA+1) is available without assigning it.

If virtualDA is eofDA, AssignDiskPage makes a free-choice assignment. The disk object remembers the virtual DA of the last page assigned and uses it as the first page to attempt to assign next time AssignDiskPage is called with a virtualDA of eofDA. This means that you can force a file to be created starting at a particular virtual address by means of the following strategy:

```
ReleaseDiskPage(disk, AssignDiskPage(disk, desiredVDA-1))
CreateDiskFile(disk, ...)  // or whatever (e.g., OpenFile)
```

ReleaseDiskPage(disk, virtualDA) marks the page as free in the bit table. It does not write anything on the disk (but DeleteDiskPages does).

VirtualDiskDA(disk, lvRealDA) returns the virtual disk address, given a real disk address in rv lvRealDA.

(The address, lvRealDA, is passed because a real disk address may occupy more than 1 word.) This procedure returns eofDA if the real disk address is zero (end-of-file), and fillInDA if the real disk address does not correspond to a legal virtual disk address in this file system.

RealDiskDA(disk, virtualDA, lvRealDA) computes the real disk address and stores it in rv lvRealDA. The function returns true if the virtual disk address is legal, i.e. within the bounds of disk addresses for the given "disk." Otherwise, it returns false.

7. Lower-level Disk Access

The transfer routines described previously have the property that all disk activity occurs during calls to the routines; the routines wait for the requested disk transfers to complete before returning. Consequently, disk transfers cannot conveniently be overlapped with computation, and the number of pages transferred consecutively at full disk speed is generally limited by the number of buffers that a caller is able to supply in a single call.

It is also possible to use the disk routines at a lower level in order to overlap transfers with computation and to transfer pages at the full speed of the disk (assuming the file is consecutively allocated on the disk and the amount of computation per page is kept relatively small). The necessary generic disk operations and other information are available to permit callers to operate the low-level disk routines in a device-independent fashion for most applications.

This level makes used of a Command Block Zone (CBZ), part of whose structure is public and defined in Disks.d, and the rest of which is private to the implementation. The general idea is that a CBZ is set up with empty disk command blocks in it. A free block is obtained from the CBZ with GetDiskCb and sent to the disk with DoDiskCommand. When it is sent to the disk, it is also put on the queue which GetDiskCb uses, but GetDiskCb waits until the disk is done with the command before returning it, and also checks for errors.

If you plan to use these routines, read the code for ActOnDiskPages to find out how they are intended to be called. An example of use of these routines in a disk-independent fashion (i.e., using only the public definitions in Disks.d) may be found in the DiskStreamsScan module of the Operating System. Only in unusual applications should it be necessary to make use of the implementation-dependent information in Bfs.d or Tfs.d.

InitializeDiskCBZ(disk, cbz, firstPage, length, retry, lvErrorRoutine). CBZ is the address of a block of length words which can be used to store CBs. It takes at least three CBs to run the disk at full speed; the disk object contains the values DSK.lengthCBZ (fixed overhead) and DSK.lengthCB (size of each command block) which may be used to compute the required length (that is, length should be at least lengthCBZ+3*lengthCB). FirstPage is used to initialize the currentPage field of the cbz. Retry is a label used for an error return, as described below. lvErrorRoutine is an error routine for unrecoverable errors, described below; it defaults to a routine that simply invokes SysErr. The arguments after firstPage can be omitted if an existing CBZ is being reinitialized, and they will remain unchanged from the previous initialization.

cb = GetDiskCb(disk, cbz, dontClear[false], returnIfNoCB[false]) returns the next CB for the CBZ. If the next CB is empty (i.e., it has never been passed to DoDiskCommand), GetDiskCb simply zeroes it and returns it. However, if the next CB is still on the disk command queue, GetDiskCb waits until the disk has finished with it. Before returning a CB, GetDiskCb checks for errors, and handles them as described below. If there is no error, GetDiskCb updates the nextDA and currentNumChars cells in the CBZ, then calls cbz>>CBZ.cleanupRoutine(disk, cb, cbz). Next, unless dontClear is true, the CB is zeroed. Finally, the CB is returned as the value of GetDiskCb. If returnIfNoCB is true, GetDiskCb returns zero if there are no CBs in the CBZ or the next CB is still on the disk command queue.

If the next CB has suffered an error, then GetDiskCb instead takes the following actions. First it increments cbz>>CBZ.errorCount. If this number is ge the value disk>>DSK.retryCount, GetDiskCb calls the error routine which was passed to InitializeDiskCBZ; the way this is done is explained in the description of ActOnDiskPages above. (If the error routine returns, GetDiskCb will proceed as if an error hadn't occurred.) Otherwise, after doing a restore on the disk if errorCount ge disk>>DSK.retryCount/2, it reinitializes the CBZ with firstPage equal to the page with the error, and returns to cbz>>CBZ.retry (which was initialized by InitializeDiskCBZ) instead of returning normally. The idea is that the code following the retry label will retry all the incomplete commands, starting with the one whose page number is cbz>>CBZ.currentPage and whose disk address is cbz>>CBZ.errorDA.

DoDiskCommand(disk, cb, CA, DA, filePtr, pageNumber, action, nextCb) Constructs a disk command in cb with data address CA, virtual disk address DA, serial and version number taken from the virtual file id in filePtr, page number taken from pageNumber, and disk command specified by action. The nextCb argument is optional; if supplied and nonzero, DoDiskCommand will "chain" the current CB's label address to nextCb, in such a way that the DL.next word will fall into nextCb>>CB.diskAddress.

DoDiskCommand expects the cb to be zeroed, except that the following fields may be preset; if they are zero the indicated default is supplied:

labelAddress	lv cb>>CB.label
numChars	0

If DA eq fillInDA, the real disk address in the command is not set (the caller should have either set it explicitly or passed the CB as the nextCb argument for a previous command). Actions are checked for legality.

The public cells in the CBZ most likely to be of interest are the following:

client: information of the caller's choosing (e.g., a pointer to a related higher-level data structure such as a stream.)

cleanupRoutine: the cleanup routine called by GetDiskCb (defaulted to Noop by InitializeDiskCBZ).

currentPage: set to the firstPage argument of InitializeDiskCBZ and not touched by the other routines. (Note, however, that GetDiskCb calls InitializeDiskCBZ when a retry is about to occur, so when control arrives at the retry label, currentPage will be set to the page number of the command that suffered the error.)

errorDA: set by GetDiskCb to the virtual disk address of the command that suffered an error.

nextDA: set by GetDiskCb to the virtual disk address of the page following the one whose CB is being returned. (This information is obtained from the next pointer in the current page's label. Note that errorDA and nextDA are actually the same cell, but they are used in non-conflicting circumstances.)

currentNumChars: set by GetDiskCb to the numChars of the page whose CB is being returned.

head: points to the first CB on GetDiskCb's queue; contains zero if the queue is empty.

8. Error Codes

The following errors are generated by the BFS. Similar errors are generated by other instances of a disk object.

1101	unrecoverable disk error
1102	disk full
1103	bad disk action
1104	control block queues fouled up
1105	attempt to create a file without creation ability

1106 can't create an essential file during NewDisk
1107 bit table problem during NewDisk
1108 attempt to access nonexistant bit table page

9. Implementation -- Bfs

The implementation expects a structure BFSDSK to be passed as the "disk" argument to the routines. The initial portion of this structure is the standard DSK structure followed by a copy of the DiskDescriptor header and finally some private instance data for the disk in use. (Note: The Alto operating system maintains a static sysDisk that points to such a structure for disk drive 0.)

Bfs ("Basic File System") is the name for a package of routines that implement the disk class for the standard Alto disks (either Diablo Model 31 drives or a single Diablo Model 44 drive). The definitions (in addition to those in AltoFileSys.D and Disks.D) are contained in Bfs.D. The code comes in two "levels:" a "base" for reading and writing existing files (implements ActOnDiskPages, RealDiskDA and VirtualDiskDA only); and a "write" level for creating, deleting, lengthening and shortening files (implements WriteDiskPages, CreateDiskFile, DeleteDiskPages, AssignDiskPage, ReleaseDiskPage). The source files BfsBase.Bcpl, Dvec.Bcpl and BfsMl.Asm comprise the base level; files BfsWrite.Bcpl BfsCreate.bcpl, BfsClose.bcpl, and BfsDDMgr.bcpl implement the write level.

BfsMakeFpFromLabel(fp, la) constructs a virtual file id in the file pointer fp from the real file id in the label la.

disk = BFSInit(diskZone, allocate[false], driveNumber[0], ddMgr[0], freshDisk[false], tempZone[diskZone]) returns a disk object for driveNumber or zero. The permanent data structures for the disk are allocated from diskZone; temporary free storage needed during the initialization process is allocated from tempZone. If allocate is true, the machinery for allocating and deallocating disk space is enabled. If it is enabled, a small DDMgr object and a 256 word buffer will be extracted from diskZone in order to buffer the bit table. A single DDMgr, created by calling 'ddMgr = CreateDDMgr(zone)', can manage both disks. If freshDisk is true, BFSInit does not attempt to open and read the DiskDescriptor file. This operation is essential for creating a virgin file system.

success = BFSNewDisk(zone, driveNum[0], nDisks[number spinning], nTracks[physical size], dirLen[3000], nSectors[physical size]) creates a virgin Alto file system on the specified drive and returns true if successful. The zone must be capable of supplying about 1000 words of storage. The logical size of the file system may be different from the physical size of driveNum: it may span both disks (a 'double-disk file system'), or it may occupy fewer tracks (a model 44 used as a model 31). The length in words of SysDir, the master directory, is specified by dirLen. Some machines that emulate Altos implement 14 sectors per track.

BFSExtendDisk(zone, disk, nDisks, nTracks) extends (i.e. adds pages to) the filesystem on 'disk'. Presumably 'nDisks' or 'nTracks' or both is bigger than the corresponding parameters currently in disk. A single model 31 may be extended to a double model 31 or a single model 44 or a double model 44, and a single model 44 may be extended to a double model 44. The zone must be capable of supplying about 750 words of storage.

0 = BFSClose(disk, dontFree[false]) destroys the disk object in an orderly way. If dontFree is true, the ddMgr for the disk is not destroyed; presumably it is still in use by the other disk. (Note that this procedure is the one invoked by the CloseDisk generic operation.)

BFSWriteDiskDescriptor(disk) insures that any important state saved in memory is correctly written on the disk.

virtualDA = BFSFindHole(disk, nPages) attempts to find a contiguous hole nPages long in disk. It returns the virtual disk address of the first page of a hole if successful, else -1.

BFSTryDisk(drive, track, sector[0]) returns true if a seek command to the specified track on the specified drive is successful. Note that the drive argument can contain an imbedded partition number. Seeks to track zero will fail if the drive is not on line. Seeks to track BFS31NTracks+1 will fail if the drive is a model 31.

10. Implementation -- Tfs

Operation and implementation of the Trident T80 disks is described in separate documentation under the heading "TFS/TFU" in Alto Subsystems documentation.

11. BFSTest

BFSTest is used to test the Basic File System (BFS) and Disk Streams software packages. It creates, deletes, reads, writes and positions files the same way that normal programs do, and checks the results which normal programs do not do. These high-level operations cause patterns of disk commands which are quite different from those generated by lower-level tests such as DiEx.

When started, BFSTest asks you which disks to test, whether to erase them first, and how many passes to run. You can use a disk with other files on it, and BFSTest will not disturb them if you prohibit erasing. The duration and throughness of a pass depends on the amount of free space on the disks.

BFSTest creates as many test files (named Test.001, Test.002, ...) as will fit on the disk, filling each file with a carefully chosen test pattern. When it is done, it deletes all of the files. One 'pass' consists of stepping through the test files, performing a randomly chosen operation on the file, and checking the results. It looks for commands from the keyboard after each file. The current commands are:

Q Quit	Delete all test files and stop.
S StopOnError	Wait until a character is typed.

All test files are 100 pages long. Each page of a file has the page number in its first and last words and a data pattern in the middle 254 words. The data pattern is constant throughout a file, consisting of a single one-bit in a word of zeros or a single zero-bit in a word of ones. Files are read and written with ReadBlock and WriteBlock using buffers whose lengths are not multiples of the page size. The operations are:

Write	Write the entire file with the data pattern.
Read	Read the entire file checking the data pattern.
Delete	Delete the file, create it again and then write it.
Copy	Copy the file to some other randomly chosen file. If both disks are being tested, one third of the time pick a destination file on the other disk.
Position	Position to twenty randomly chosen pages in the file. Check that the first word of the page is indeed the page number. One third of the time dirty the stream by writing the page number in the last word of the page.