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.