ALTO OPERATING SYSTEM
REFERENCE MANUAL

Compiled on: June 26, 1975

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

Alto Operating System Reference Manual
Version 1

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 in this manual) can be separated into three 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. At the moment, all procedures are standard except those for parsing command files (GP); procedures for doing page-oriented disk I/O are also available as library procedures (BFS).

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 currently 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 48k or 64k words of 16 bits each, plus parity. The cycle time is 850ns.

b) An emulator for the Nova instruction set, except the input/output instructions (which include MUL, DIV, HLT and the instructions which control the interrupt system). The only other incompatibilities are:
   1) Addresses are 16 bits, rather than 15, so that bit 0 of an index register affects the addressing.
   2) Indirect addresses are also 16 bits, so that bit 0 is part of the address, rather than specifying another level of indirection.
   3) Auto-increment and auto-decrement are not implemented.

   There are some new instructions which are listed in Table 2.1. The Alto executes emulated instructions in about 1.5 times the time required by the Nova 800: about 1.2 us for register instructions, 2 us for loads and stores.

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. The formats of the disk and of disk commands are described in section 4.2.

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 whose format is described in section 4.5. 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. At the moment, all of these procedures are a permanent part of the operating system, automatically available to any user program.

3.1. Initialization

Because of the deficiencies of the Bcpl loader, it is necessary to use a kludge to link the user program to the operating system routines. This kludge is embodied in a routine called INITALTOIO, which must be loaded with the user program, and must be called before any operating system procedure is called. This routine finds out where the system routines are and plants their addresses in the statics through which the user program references them. Statics which are set up by INITALTOIO are called system procedures in this document, or system scalars if they aren't procedures. Most system scalars contain the addresses of locations in the system which contain interesting information, so that if SC is a system scalar, rv SC will be the proper expression to use in a program.

INITALTOIO takes an optional parameter, which is the maximum number of disk streams which will exist during execution of the user program - 3. It may be omitted if the user program needs no more than 3 disk streams. In any case it should not be larger than 4; if your program needs more that five disk streams, consult me. The need for this parameter will disappear shortly.

The space occupied by INITALTOIO can be reclaimed after it has been called by loading it at the end of your program, and doing SetEndCode(INITALTOIO) after calling it. Multiple calls of the routine are harmless provided they are parameterless (unless, of course, you have smashed it as described in the last sentence).

3.2. Errors

There is a standard routine, SYSERR, which is called whenever the system detects an error from which it doesn't know how to recover, and the user program has not supplied its own error routine. It takes two parameters:

N: a number whose significance depends on the kind of error;
EC: an error code, which can be interpreted by looking it up in Table 3.1.

SYSERR simply types N (in octal), followed by EC, the error code, which unfortunately comes out in octal. After doing its typing, it says

Type F to finish

and waits for inputs. Typing F (or f) will do a finish. Typing two successive control-R's will do a return (normally not a good idea). Anything else will be echoed and ignored. The reason for this pause is to provide an opportunity for invoking Swat.

In this document errors are described as "fatal", meaning that a return from SYSERR is useless, or "non-fatal," meaning that something reasonable will happen if SYSERR returns.

There is also a routine called SWAT(message) which just types the message and then says "Type F ..." as above.

3.3. Streams

The purpose of streams is to provide a standard interface between programs and their sources of sequential input and sinks for sequential output. There is a set of standard operations, defined for all streams, which are 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, but the 16-bit quantity which Bcpl passes as an argument or receives as a result could be a pointer to some larger object such as a string. In this case, the storage allocation conventions for the objects thus transmitted would have to be defined. 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 user is free to construct his own streams by setting up a suitable data structure (defined in section 4.1) which provides links to his own procedures which implement the standard operations.

The system should take precautions to ensure that its streams will be destroyed in an orderly way when execution of a program is complete, in case the program has forgotten to CLOSE them, but unfortunately at present it does not, and programs should therefore take care to close all their streams before finishing. The routine CLOSEALL will do this automatically.

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; error if ENDOFS(S) is true before the call
PUTS(S,I)	writes I into the stream as the next item; error only if there is no more space or some hardware problem.
RESETS(S)	restores the stream to some initial state, generally as close as possible to the state is is in just after it is created.
PUTBACK(S,I)	modifies S so that the next GETS(S) will return I and leaves S in the state it was in before the PUTBACK. Error if there is already a putback in force on S.
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. ERRORS is initialized to SYSERR (see section 3.2), but the user can replace it with his own error routine.
OPENS(S)	for display and EIA-input, turns the stream on; for other types, does nothing. This operation will be abolished shortly.

Currently, a stream is created by a call of CREATES(P, TYPE, ERR), which returns the new stream. ERR is the error routine; if it is 0 or missing, SYSERR is used. P is a parameter whose interpretation depends on the type. The possible types are listed in Table 3.2; these symbols are defined in the file OSSYMS. Properties which are specific to each type are described in the following subsections.

3.3.1. Disk streams

A disk stream is the standard interface to a disk file. Section 3.4 describes how to obtain a stream for doing input or output to a particular file. This section describes operations peculiar to disk streams.

A disk file is a sequence of bytes numbered from 0 to N; it is divided into pages of 512 bytes which are numbered starting at 1. The length of the file is N+1. A disk stream is always positioned at some byte P of the file; if the items are words, P will always be even.

When it is created, a disk stream is positioned at the beginning of the file, i.e., P=0. RESETS repositions it to the beginning. GETS returns byte P and sets P=P+1 (for a word stream, it returns bytes P and P+1, and sets P=P+2), unless P>N, in which case it calls ERRORS. PUTS writes its argument into byte P and sets P=P+1

(and accordingly for a word stream); it also sets N to max(N,P). ENDOFS returns true if P>N. CLOSES updates the disk image of the file, if necessary, and destroys the stream; it does not truncate a file. This action is performed by CLOSEAFILE (see section 3.4) or DELETEFILES(stream, pagenumber, bytenumber), which sets N to the specified byte (note that data pages are numbered starting at 1, so byte 514 is page 2, byte 2). If pagenumber is 0 or missing, the file associated with the stream is entirely deleted. For this and other uninteresting reasons, it is better not to use CLOSES on disk streams in the current system. This peculiarity will be remedied soon.

There are several functions which deal with file positions:

POSITIONPAGE(S, N)	positions to byte 0 of page N. The disk file is lengthened if necessary to make this possible, and filled in with garbage. Don't forget that the first data page is numbered 1.
POSITIONPTR(S, N)	positions to byte N-2 (sorry about that) of the current page, lengthening the file if necessary. Do not position word streams to odd bytes.
MOVESTREAM(S, N)	positions to byte (P - P mod 2)+2*N, where P is the current position and N is signed, using the two previous functions.
FilePos(S, vector)	returns the current byte position in the stream, modulo 2**16. If vector is supplied, it also returns the position in double-precision in vector!0 (msb) and vector!1 (lsb).
FileLength(S, vector)	returns the length in bytes of the file associated with the stream, modulo 2**16. If vector is supplied, it also returns the length in double-precision in vector!0 and vector!1. It positions the file to the end.

Finally, there are two functions for block transfers. Do not use them if the current position is not even.

READVEC(S, addr, count)	reads max(count+1, number of words remaining) from S into memory, putting the first word at addr. Returns the number of words read - 1. If the length of the file is odd, the last byte will be read, together with a garbage byte. READVEC is equivalent to: for i=0 to count do addr!i=GETS(S) if S doesn't run out, but it is much more efficient, running at full disk speed except, perhaps, for the first and last pages.
WRITEVEC(S, addr, count)	writes count+1 words into S from memory, taking the first word from addr. The current version can only write one page per two disk revolutions, or one page per four disk revolutions if the file is being extended.

Disk streams are normally created using the functions of section 3.4. It is also possible, however, to use the function CREATES with a disk stream type and P the address of a file identifier (see section 3.4).

Each disk file has a leader page associated with it which contains various information. The first six words of the leader page contain (in order) the time of creation, time of last write, and time of last read of the file. (These times are obtained from the DAYTIME procedure, section 3.7.) Each time a file is opened, the mode in which it is opened is used to update these times: if a file is opened read/write (the default) both read and write times are updated. Therefore, try to specify exactly what you want to do when you do the open.

ReadFileStuff(stream,v) Read the leader page of the file open on "stream" into the vector v (which had better have room for 256 words). Currently the only interesting information in the leader page is the time stamps.

3.3.2. String streams

A standard Bcpl string can be made into a stream by CREATES(string, STRINGOPEN). It then behaves much like a disk stream, except that ERRORS is called if any attempt is made to extend its length beyond 255 bytes (in accordance with the Bcpl convention, the length is stored in the left byte of string!0, and the first byte is in the right half). The disk stream operations DELETEFILES, POSITIONPAGE, READVEC and WRITEVEC do not work on string streams, but everything else is identical.

To handle big strings, use CREATES(stringvec, BIGSTRINGOPEN), where stringvec!0 contains the length, and stringvec!1 contains the first two bytes, etc. Big string streams work exactly like string streams except that the length restriction is 65k bytes instead of 255.

3.3.3. Display streams

This section describes the current facilities for managing the display screen and putting text on it. These facilities will be substantially changed in the near future.

There is a standard font which the system uses for text, unless the user specifies another one. In this font, a full text line contains 72 characters and requires 508 words of memory . The characters are 14 dots high and 8 dots wide, including the surrounding space.

The system takes 6 text lines (84 scan lines) at the top of the screen. This area can be accessed through a stream which can be found in the system scalar DSP.

The user can create his own display streams with CREATES(V,DISPLAYOPEN). Once a display stream S is created, it can be appended to the existing display by OPENS(S), and removed by CLOSES(S).

The parameter V is interpreted as follows:

a) V=0 or V!0=0 or V!1=0: the system display stream DSP is returned. This is for compatibility with earlier versions of the system.

b) otherwise: V should be a 4-word vector. V!0 and V!1 are taken as the starting address and ending address of a region of memory which is allocated for the display. Independently, if V!2 is non-zero, it is taken as a font pointer (see section 4.5), and V!3 as the number of words to allocate for each scan line (must be even); the number of characters which will fit on a line (for a fixed-pitch font) is then 16*V!3/width of a character in dots.

3.3.4. Keyboard

There is a single keyboard stream in which characters are buffered. It can be obtained by doing CREATES(0,KEYSOPEN), and can also be found in 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.

There are currently no system facilities for handling the keyset or mouse, except for the system scalar MOUSELINK, which has the property that the cursor coordinates will follow the mouse coordinates iff rv MOUSELINK is not false.

3.4. Disk files

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

stream:	used by a program to transfer information to or from a disk file; .
file:	a vector of bytes of data held on the disk, organized into 512-byte pages for some purposes;
directory:	a file which contains a list of pairs <string name, file>.

A stream exists only in memory, is defined by a structure described in section 4.1 and is named by a pointer to this structure. 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 a 64-bit entity which consists of a 32-bit serial number, a 16-bit version number, and a 16-bit disk address of the leader page for the file (this is a virtual disk address; see section 4.4). This name, together with some other information, is packaged by the system into a data structure in memory, called a file identifier (section 4.2).

Most user programs do not concern themselves with file identifiers, but use system routines which go directly from string names to streams. These routines will be described first, and the interested reader can obtain more details further on.

3.4.1. File names

By a "file name" we mean 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.

3.4.2. Functions for the naive user

The current system supports a single directory. It is always available as a stream which can be accessed from the system scalar SYSTEMDIR. All the operations in this section look things up in this directory. The directory is a file, and the file appears in the directory itself under the name SYSDIR.

GETAFILE(string, type [, error routine]) looks up the string in the directory, and creates a new file and a new directory entry if

it doesn't find the string there. Then it creates a stream for the file, with the specified type (see Table 3.2 and section 3.3) and returns it. The type defaults to read/write words (DISKRW). The optional third parameter is used to set the stream's ERRORS; if it is missing, SYSERR is used.

OPENAFILE(string, type [, error routine]) is identical to GETAFILE, except that it will return 0 if it doesn't find the name in the directory; i.e., it will not create a new file.

CLOSEAFILE(stream) will destroy the stream in an orderly way, writing onto the disk any data still in memory and releasing all the memory occupied by the stream. If the stream was created write-only, the file will be truncated at the current position, unless it is positioned at the beginning.

DELETEAFILE(string) deletes the file and removes the directory entry.

The naive user should consult sections 3.3 and 3.3.1 for information about how to use the streams created by GETAFILE and OPENAFILE.

3.4.3. Lower-level directory functions

As was explained in the last section, the current system supports a single directory file, which is always kept open as a stream, the system scalar SYSTEMDIR. This file has its leader page in disk page 1, and appears in itself under the name SYSDIR. The routines of the last section all work on SYSDIR.

The routines in this section, on the other hand, will work on any stream which has the format of a directory: a sequence of entries, each one a data structure described in section 4.3. These routines all take a parameter DIR which is a stream of type DISKRW for the file to be used as the directory. The parameter NAME is a Bcpl string.

LOOKUPENTRY(DIR,NAME) looks up NAME in DIR. If it is found, DIR is left positioned at the first word of the entry and true is returned; otherwise, false is returned.

FINDHOLE(DIR,N) leaves DIR positioned at the first word of a free space (or hole) at least N words long. The value is the size of the hole, or 0 if it is at the end of the file.

MAKENTRY(DIR,NAME,TYPE) If TYPE=NILTYPE(=0), deletes the entry for NAME from DIR; non-fatal error if no such entry exists. Note that the file is not deleted, only the entry. If TYPE=REALTYPE(=1), makes a new file with the specified name and enters it on DIR; non-fatal error if an entry for NAME already exists. In both cases, there is a non-fatal error if the NAME contains a character which is forbidden in file names. Fatal error if TYPE is not one of these cases.

There is currently no routine to make a directory entry for an existing file, but it is easy to program. Don't forget that if FINDHOLE returns a non-zero value which isn't the right size, you must write an appropriate hole entry after the new entry you make.

3.4.4. 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 of ERRORS for the stream involved. The rv of system scalar REXQTCOUNT contains the number of times a disk operation has been repeated because of an error.

3.5. Memory management

Table 3.3 shows the layout of memory . Table 3.4 tells how to obtain the current values of the symbolic locations in Table 3.3. 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(new value)	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 freelist. When another FreeFixed is done, any blocks on the freelist which are now just below endCode will also be freed. However, the allocator makes no attempt to allocate blocks from the freelist.

The system is initialized with 3 disk streams (plus SYSTEMDIR), 8 string streams, 1 keyboard stream, and 2 display streams. The number of disk streams can be increased (once) by MOREFILES(N), where N is the number of extra streams; it must be le 4. The space is allocated inside the system, and this action is undone by a FINISH. If INITALTOIO is given a parameter N, it calls MOREFILES(N). If absolutely necessary, the number of other objects can be increased in an undocumented way.

3.6. Subsystems and user programs

The information in this section is likely to become invalid in the next few months. Try to isolate any use you make of it in your programs.

All subsystems and user programs are stored as save files, which normally have no extension. 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 save file is specified in section 4.7 and table 4.11.

To swap in a save file and send control to its starting address, do OVERWRITE(S [,swatflag]), where S is a DISKRO or DISKRW stream for the file, positioned at the beginning of the file. Swatflag is an optional flag, which if true causes SWAT to be called just before control is given to the program being called. The new program

gets control with the system in a clean state (no streams open except SYSTEMDIR, no user stack). OVERWRITE puts the statics and the code where SV.BLV.startOfStatics and SV.BLV.startOfCode say they should go. It checks that the OS and the stack will not be overwritten.

The program will be started by a call to its starting address (SV.H.startingAddress), which will normally be the first function of the first file given to Bldr. This function is passed a single argument, which is a 32 word layout vector described in Table 4.10 (taken from the Bcpl manual). When it returns, either by doing a return from the function which was first called, or by doing a FINISH, the system is reinitialized and control goes back to the command processor.

Don't forget that INITALTOIO must be called before any calls on the operating system are made by a newly invoked program.

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 (copied from Nova DOS) 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 PACKAGES documentation.)

3.7. Miscellaneous

This section describes a collection of miscellaneous useful routines:

WS(string)	writes the string on the standard output system (currently always the system display)
WO(n)	writes an unsigned octal representation of n on the standard output stream
WSS(S, string)	writes the string on stream S
WOS(S, n)	writes an unsigned octal representation of n on stream S
BSTORE(addr, v, c)	stores v into addr!0...addr!c (note: c+1 stores are done)
BMOVE(source, dest, c)	does dest!0←source!0...dest!c←source!c (note: c+1 moves are done)
TIMER(tv)	Reads the 32-bit millisecond timer into tv!0 and tv!1. Returns tv!1 as its value.
DAYTIME(dv)	Reads the current time-of-day (32 bits, with a grain of 1 second) into dv!0 and dv!1. Returns dv as its value. (Subroutines for converting time-of-day into more useful formats for human consumption are available in CTIME.C. See subroutine package documentation, under TIME.)
SETDAYTIME(dv)	Sets the current time-of-day 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.)

4. Data structures

This section describes the data structures used by the operating system. Each structure is described by a Bcpl structure declaration, if appropriate, together with explanatory text. All the structure declarations can be found in the file OSSTRUCTURES.

Table 4.1 lists all the addresses in page 1 which are used by the standard Alto.

4.1. Streams

Table 4.2 gives the minimal structure for a stream. Any particular stream structure may be larger. The standard entries are all Bcpl procedures. Blank entries are type-dependent. The type field is the parameter given to CREATES for a system-provided stream.

4.2. Disk files

A sector on the disk contains a 2-word header, an 8-word label and a 256-word data record, each with a checksum. Table 4.3 gives the structures of the header and the label, and also defines the subsidiary structures of a disk address and a serial number. The DH.packId field is currently always 0, but will eventually be used to identify the pack. The SV.random field is currently always 0, but may someday be used to specify that information about the disk addresses of the file pages is recorded on the disk.

Disk activity is specified by disk command blocks. Table 4.4 gives the structure of a DCB and also of a disk command and a status word.

A file is named by a file identifier whose structure is given in Table 4.5. The FID.fileNumber field is currently unused. The FID.name field is also currently unused, but it will be a Bcpl string which is the name which was used to access the file. Its maximum size is 128 words, although no more space need be allocated than is actually required for the name in any specific case.

4.3. Directories

A directory file is a sequence of directory entries, each with the structure specified in Table 4.6. It should have the SN.directory bit set in its serial number. Note that it is possible to have a sequence of free entries, since they are not coalesced until FINDHOLE tries to obtain space. Note also that free entries may be of any size.

4.4. Disk allocation

Free space on the disk is kept track of in two ways. First, each free block has a label which is zero, except for the DL.serialNumber and versionNumber fields, which are all ones. Second, there is a bit table which contains a bit for each disk page: the bit is 0 if the page is free, 1 if it is allocated. Whenever the system allocates disk space, it uses the bit table to find a free page and checks the label to make sure it is free, and then immediately rewrites the label to reflect its newly allocated status. The bit table is indexed by virtual disk address; the leftmost bit in a word is bit 0. The virtual address for a sector with real address a is:

```
a<<DA.sector + 12 * a<<DA.head + 24 * a<<DA.track
```

There are two system functions for conversion: VIRTUALADDRESS(real address) and MAKEADDR(virtual address).

The bit table is stored on a file called SYS.STAT, followed by two words containing the largest serial number so far. It is also kept in core in a vector which may be found in the system scalar BITTABLE. The disk copy is updated whenever a subsystem returns.

4.5. Display

The display is defined by a list of control blocks, each of which specifies a band or contiguous group of scan lines. Table 4.7 gives the format of a display band control block. DBchainHead in page 1 points to the first DB; if it is 0 the display will be off.

There is a standard format for Alto fonts, which has been chosen to be compatible with the scan conversion instruction; it is given in Table 4.8. The font pointer passed to CREATES must point to the characterFCDpointers word of the font.

4.6. The Bcpl stack

The Bcpl compiler determines the format of a frame, which is given in Table 4.9, and the calling convention. The strategy for allocating 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
L:      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 (see section 3.5). To allocate a new frame of size S, it simply computes NF=AC2-S and checks to see whether NF > EndCode. If not, there is a fatal error; if so, NF becomes the new frame.

4.7. Save files

Table 4.11 gives the format of an Alto save file. The system routine OVERWRITE (section 3.6) will read such a file into core and start it running. Someday this file will have a variant which includes a saved stack, so that control can return from OVERWRITE to its caller by saving the caller's state in such a file.

5. The command processor

The Alto command processor is itself a subsystem and lives on the file COMMAND; 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 command processor is fully documented with SUBSYSTEMS; only a brief description is presented here.

The command processor reads a command line from the keyboard, writes it (with some interpretation) onto the file COM.CM, terminated with a carriage return, and calls in the file named by the first word on the line (up to blank, / or carriage return). The interpretation is as follows:

a) If more than one display line is needed, a command line may be continued on the next display line by preceding the carriage return with a ↑. This ↑ simply causes the carriage return to be ignored; it does not act as a separator. A ↑ not followed by carriage return is treated as an ordinary character. Line-feed characters are ignored.

b) If the sequence @filename@ appears, the contents of the specified file are treated as though they had been typed in at that point, instead of the @ construction. This may be nested to any reasonable depth.

c) The backspace key, or a control-A, deletes the previous undeleted character; a DEL deletes the whole line. A control-R retypes the line. Two slashes (//) begin a comment, which is terminated by the carriage return or semi-colon which terminates the command.

d) Commands can be separated by semi-colons. If there is more than one command in a command line, everything following the first command is saved (after the interpretation described above) on a file called REM.CM, and a password is stored into a location called REMCM in the operating system. When the commmand processor is invoked, it checks REMCM for the password and simulates the typing of @REM.CM@ if it is there.

The command processor has some simple commands built into it, rather than accessed through the subsystem machinery. Currently these are DELETE and TYPE. They are handled exactly like other commands, but are somewhat more efficient, and cannot be overridden by changing the save files with those names.

6. Operating Procedures

6.1. Getting started

To get started on the Alto, obtain an Alto disk pack. It must be a twelve-sector pack; i.e. it should say 902-12: H.D. on the front. Label the pack with your name and other suitable identification, since unlabeled packs are liable to garbage collection.

Find the pack labeled BASIC ALTO DISK, and put it into the bottom disk drive on an Alto which has two drives. Push the white switch on the drive to RUN. When the yellow READY light is lit, push the Alto's boot button, which you will find on the rear of the keyboard. The screen should say

```
Hello, Basic Alto disk
  some information about the status of the machine
>
```

You are now talking to the command processor. Type

```
QUICK cr
```

When it asks you if you are ready to proceed, move the BASIC ALTO DISK to the top drive and put your new pack into the bottom drive. Then say yes, and when it asks you whether you are really ready, be sure that you have the disks in the right drives (new disk in the bottom drive, BASIC disk in the top drive) and say P. The program will copy the BASIC disk onto your new disk and will then read back the new disk and compare it with the BASIC disk. It will tell you when it is done. Take out the BASIC disk and put it back in the rack. You can now boot from your new disk, and you are ready to go. It is probably a good idea to start by INSTALLing the system with your name in it; INSTALL is described in the subsystems documentation.

6.2. Miscellaneous information

The key in the lower right corner of the keyboard is called the Swat key. If you press it, the Swat debugger will be invoked. If you do this by mistake, control-P will resume your program without interfering with its execution, and control-K will do a finish.

You can force a finish at any time by depressing the Swat key together with the left-hand shift key.

Name	Opcode	Address	Function
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.
CYCLE	60000	C	AC0←AC0 lcy (if C ne 0 then C else AC1)
JSRII	64400	D	AC3←PC+1; PC←rv (rv (PC+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 last 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 last 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)

Notes: Address: C=bits 12-15; D=bits 8-15; -=no address
variables in function descriptions are machine registers or page 1 locations (see table 4.1)

Table 2.1: New instructions in Alto emulator
(see Hardware Manual, section 3.1 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

0: Unspecified error
1: Bad stream
2: No such action
3: There is no more room for streams
4: Parameter too small
5: Parameter too big
6: Bad parameter
7: End of stream
8: Bad PUT
9: Bad GET
10: PUTBACK attempted when there is already a put back item
11: Hardware I/O error
12: Bad state
13: Bad name
14: No such entry in the directory
15: Bad file
16: Not yet implemented
17: Page not full
18: Page number too small
19: Bad disk address
20: File already exists
21: Too many objects
200: Attempt to overlay stack
201: Stack is too big
202: No more stack space
203: Statics overlap stack or system
204: Code overlaps stack or system
220: Interrupt address already exists
221: Interrupt channel already active

Table 3.1 System error messages; see section 3.2

Type name	Numeric value	Section	Device	Item	Remarks
DISKRO	0	3.3.1	disk	word	read-only
DISKWO	1	3.3.1	disk	word	write-only truncated at current position by CLOSEAFILE
DISKRW	2	3.3.1	disk	word	read-write
DISKROCH	3	3.3.1	disk	byte	read-only
DISKWOCH	4	3.3.1	disk	byte	write-only truncated at current position by CLOSEAFILE
DISKRWCH	5	3.3.1	disk	byte	read-write
STRINGOPEN	6	3.3.2	Bcpl string	byte	
BIGSTRINGOPEN	7	3.3.2	'big' string	byte	
KEYSOPEN	9	3.3.4	keyboard	byte	
DISPLAYOPEN	10	3.3.3	display	byte	for text only

Table 3.2: Stream types; see section 3.3

LastMemLoc	Last memory location
...	
StartSystem	Base of system
...	
StackBase	Root of stack
...	This region is permanently used by the system. Don't mess with it.
UserStackBase	Root of user's stack
...	
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 Bldr manual.
StartCodeArea	Start of user program area
...	
1000	Start of Bcpl runtime and system machine-language routines
...	
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 3.3: Memory layout (all numbers octal); see section 3.6

LastMemLoc	See text
StackBase	rv(#1017)
UserStackBase	System won't provide this, but user's initial routine can compute it
StackEnd	lv (first argument of current procedure) -4
EndCode	Rv(#335)
StartCodeArea	Fixed at #6500

Table 3.4: Values of symbolic locations in Table 3.3
(all numbers octal)

```
// Page 1 reserved locations - from Alto hardware manual, Appendix B
// and passim

manifest [

// display
        DBchainHead = #420              // pointer to DB
        verticalFieldInterrupt = #421   // 60 times a second

// cursor
        cursorX = #426
        cursorY = #427
        cursorBitMap = #431             // up to #450

// mouse
        mouseX = #424                   // hardware increments or
        mouseY = #425                   // decrements these words

// disk
        nextDiskCommand = #521          // pointer to DCB
        diskStatus = #522               // @DS
        lastDiskAddress = #523          // @DA
        sectorInterrupts = #524

// interrupts
        wakeupsWaiting = #452           // bit 1 goes to
                                        // interruptvector!1
        activeInterrupts = #453
        interruptedPC = #500
        interruptVector = #500          // #501 to #517

// interval timer
        timerInterruptTime = #525// interrupt occurs when AC1
                                        // after RCLK equals this word
                                        // in the 10 msb
        timerInterrupts = #423
        timerData = #422

// miscellaneous
        realTimeClock = #430            // units are 1024/30*875
                                        // seconds or about 38 ms
        trapPC = #527
        trapVec = #530                  // up to #567
        convertMaskTable = #460         // up to #477; the convert ins-
                                        // truction requires that
                                        // convertMaskTable!n=
                                        // (2**(n+1))-1

        ]



//      Table 4.1: Page 1 reserved locations
```

```
// minimal structure for a stream
structure ST[
        blank word
        open word
        close word
        gets word
        puts word
        reset word
        putback word
        error word
        endof word
        stateof word
        blank word
        blank word
        blank word
        type word
        ]
manifest lST=size ST/16
```

// Table 4.2 Basic stream structures; see section 4.1

```
// disk address
structure DA[
        sector bit 4
        track bit 9
        head bit 1
        disk bit 1
        restore bit 1
        ]

// disk header
structure DH[
        packId word
        diskAddress @DA
        ]

// serial number
structure SN[
                [
                directory bit 1
                random bit 1
                part1 bit 14
                ]
        = word1 word
        part2 word
        ]

// disk label
structure DL[
        next word                       // disk address of next file
                                        // page, or 0
        previous word                   // disk address of previous
                                        // file page, or 0
        blank word
        numberOfCharacters word         // between 0 and 512 inclusive.
                                        // ne 512 only on last page
        page word                       // leader is page 0, first data
                                        // page is page 1
        fileId word 3 =
                [
                version word
                serialNumber @SN
                ]
        ]
manifest lDL=size DL/16
```

// Table 4.3: Disk format; see section 4.2

```
// disk status word.  See hardware manual for detailed definitions
structure DS[
        sector bit 4
        done bit 4
        seekFailed bit
        seekInProgress bit
        notReady bit
        dataLate bit
        noTransfer bit
        checksumError bit
        finalStatus bit 2
        ]

// disk command
structure DC[
        seal bit 8                  // must be #110
        headerAction bit 2
        labelAction bit 2
        dataAction bit 2
        seekOnly bit 1
        exchangeDisks bit 1                 // disk controller inverts
                                            // DCB.diskAddress if this bit
                                            // is set
        ]
// possible disk actions in command word
manifest [ diskRead = 0; diskCheck = 1; diskWrite = 2]

// disk command block
structure DCB[
        nextComm    word
        status @DS                          // set when command is completed
        command @DC
        headerAddress word                  // these are memory addresses
        labelAddress word
        dataAddress word
        noErrorInterrupts word
        errorInterrupts word
        header @DH =
                [
                blank word
                diskAddress @DA
                ]
        ]
manifest lDCB=size DCB/16

//      Table 4.4: Disk commands, see section 4.2
```

```
// Bcpl string
structure STRING[
        length byte
        body  0,255 byte
        ]

// file identifier
structure FID[
        serialNumber @SN
        version word
        fileNumber word
        diskAddress @DA                 // virtual disk address
        name @STRING
        ]
```

// Table 4.5: File identifier; see section 4.2

```
// directory entry
structure DE[
        type bit 6
        nwords bit 10                   // number of words in the entry
        fid @FID                //WARNING! the DA here is virtual
        ]
// directory entry types
manifest [ DEfree = 0; DEfile = 1; DElink = 2 ]
```

// Table 4.6: Directory format; see section 4.2

```
// display band control block
structure DB[
        next word
        resolution bit 1                    // 0=high
        background bit 1                    // 0=white
        indentation bit 6                   // in units of 16 dots
        width bit 8                         // likewise; must be even
        bitMapAddress word                  // must be even
        height word                         // in double scan lines
        ]
manifest lDB=size DB/16

//      Table 4.7: Display control; see section 4.5


// font character descriptor.  If p points to a FCD, then
// p!(-p>>FCD.height) contains the first scan line of the character
// stored in the font, p!(-1) the last
structure FCD[
        [ blank bit 11; width bit 4 ] =
        [ blank bit 7; extension bit 8 ]
        noExtension bit 1           // previous field is width if
                                    // this is 1; otherwise it is
                                    // extension and the width is 16
        displacement bit 8          // character is pushed down this
                                    // far
        height bit 8                // number of scan lines (words)
                                    // of bit map stored for this
                                    // character
        ]

// font
structure FONT[
        height word                 // height of tallest character,
                                    // in scan lines, including min-
                                    // imum inter-line space. Must
                                    // be even
        variableWidth bit 1
        blank bit 7
        width bit 8                 // includes inter-character
                                    // spacing; maximum width if
                                    // font is variable-width
// The font pointer given to CREATES must point to the next word
        characterFCDpointers  0,255 word // self-relative pointers to
                                    // FCDs for 256 characters
        extensionFCDpointers word// likewise for extensions; as
                                    // many words as needed.
                                    // the FCDs follow immediately
        ]

//      Table 4.8: Standard Alto fonts; see section 4.5
```

```
// Bcpl frame
structure F[
        callersFrame word
        savedPC word                    // this frame's PC is stored
                                        // here during a call
        temp word                       // temporary storage; free
                                        // during calls
        extraArguments word             // see manual for details
        formals word                     // the formal parameters are
                                        // stored in successive words
                                        // starting here.
        ]
```

// Table 4.9: Bcpl frame structure; see section 4.6

```
// layout vector passed to the program on startup
structure BLV[
        overlayAddress  0, 25 word
        startOfStatics word                 // address of first static
        endOfStatics word           // address of last static
        startOfCode word                    // address of first word of code
        afterLastCodeWord word              // 1 + largest address at which
                                            // code is loaded (normally
                                            // endCode is the same, and the
                                            // system treats that value
                                            // as the end of the program)
        endCode word                        // first location which may be
                                            // used for data; used by the
                                            // system to set EndCode
        blank word
        ]


//      Table 4.10: Bcpl layout vector


// format of an Alto save file
structure SV[
        H [                                 // header
          startingAddress word              // initial value for PC.
                                            // Loader sets this to
                                            // SV.BLV.startOfCode
          length word                       // unused
          type word                         // should be 0 for an ordinary
                                            // save file
          blank  1, 11 word
          ]
        BLV @BLV                                    // Bcpl layout vector
        page0  0, #277 word                 // The first #16 words are
                                            // ignored; the rest are used to
                                            // set words #16 to #277 of
                                            // memory
        statics  0, 0 word          // actually there are
                                            // (BLV.endOfStatics -
                                            // BLV.startOfStatics + 1)
                                            // words here
        code  0, 0 word                     // actually there are
                                            // (BLV.endCode-BLV.startOfCode)
                                            // words here
        end word
        ]


//      Table 4.11: Alto save file
```

7. PARC Information

7.1. Getting Started

New Alto disks are stored in a cabinet in the Maxc room. Please tell Vicki Parish the serial number of any new disk that you take.

The BASIC ALTO DISK is in a rack near the Alto in the cubicle adjacent to Vicki's. The procedure for executing QUICK to copy the BASIC ALTO DISK onto another disk may be found in this manual (Operating System) under the section entitled, Operating Procedures. An alternative procedure for creating a new disk is NEWDISK. The documentation may be found in the SUBSYSTEMS manual or on <ALTODOCS>NEWDISK.TTY.

7.2. Alto Programs

Barbara Hunt maintains the following Alto directories:

1) The <ALTOSOURCE> directory. This directory contains the source files for the subsystems and subroutine packages. It also contains the PUB files for the documentation which is on <ALTODOCS>.

2) The <ALTODOCS> directory.

3) The <ALTO> directory.

Current versions of all the Alto programs are kept on Maxc on the <ALTO> directory. Programs are normally kept in executable form; thus the QUICK program appears as <ALTO>QUICK. In addition to the executable file, some programs also have a symbol file on <ALTO>. The symbol file name has the extension .BM. This file is useful to the author when something goes wrong with a subsystem, but it is not normally needed by users. Subsystems which need more than one file, either because they have overlays or because they need data files, are kept as dump files with the extension .DM. These dump files should be broken apart with LOAD after you transfer them to your Alto disk.

Subroutine packages are kept on <ALTO> with an extension of .BR.

The maintainer of a subsystem or subroutine package submits a new or revised release in a dump file which has the following contents:

1) The source files from which the subsystem may be created. The name of the file should be in the format, subsysname.DM.

2) The command files which are needed to create the subsystem from the enclosed source. The following are the usual requirements:

a) A command file containing statements to compile the enclosed source. Compiler messages should be written to a file. For example:

BCPL/F FOO.SR.

The filename should be in the format, COMPILEsubsysname.CM.

b) A command file containing statements to load the files which were produced in step a. For example:

BLDR FOO

The filename should be in the format, LOADsubsysname.CM.

If the subsystem is small, the two command files may be combined into one. The name should be in the format, CREATEsubsysname.CM.

The following example will create the package for subsystem, FOO.

DUMP FOO.DM FOO.SR CREATEFOO.CM

The command file, CREATEFOO.CM, contains the following statements:

BCPL/F FOO.SR
BLDR FOO INITALTOIO

A message should be sent to Barbara Hunt, describing the changes which will be effective with this release. Include the name of the directory on which the files may be found. The subject of the message should be the name of the subsystem or subroutine package.

7.3. Documentation

The file <ALTO>MESSAGE.TXT contains all of the information which has been sent to Alto users with SNDMSG. Information about recent changes to a specific subsystem may be selected by using the 'subject string' option of the MSG subsystem. For example, you may type
MSG <ALTO>MESSAGE.TXT
T S FOO
Or you can read the entire file by saying
File: <ALTO>MESSAGE.TXT
to READMAIL. Every six months this file will be purged and its old contents left on the next version of OLDMESSAGE.TXT.

Formal documentation is provided in two forms: a "perusal" form, which can be conveniently typed at a TI or VTS terminal on Maxc or perused with BRAVO on an Alto, and a "notebook" form, which can only be printed on ears, and may have fancy illustrations or fonts in it.

The "perusal" documentation is always stored on <ALTODOCS> under a file name like sys.TTY, where "sys" is the name of the subystem or package you are interested in. For example, the documentation for a subroutine package, FOO, would be found on <ALTODOCS>FOO.TTY. There is one exception to this rule: for very simple subsystems (e.g., DUMP and LOAD), the documentation is in <ALTODOCS>SMALLSUBSYSTEMS.TTY.

The "notebook" documentation is packaged in larger packages to reduce storage overhead and to provide more manageable sets of documentation for printing. Currently, the following files on <ALTODOCS> may be copied to lpt: for notebook-style documentation:

OS.EARS. Operating System manual. the last section of this manual contains special information relating to Altos at PARC--where to find the software, how to maintain it, etc.

BCPL.EARS. A new, revised BCPL manual.

SUBSYSTEMS.EARS. Documentation for most Alto subsystems (except those listed below). These are arranged alphabetically, with headings to indicate which system is being described. A directory at the front of the file contains documentation about very simple subsystems.

PACKAGES.EARS. This contains documentation for the software packages available for the Alto.

BRAVO.EARS, GYPSY.EARS. Currently, these subsystems have their own separate ears documentation.

APERSCOMP.EARS. This is the "hardware" manual for the Alto.

These files are formatted, and should therefore be printed with
@COPY FOO.x LPT: [confirm] ('x' is either TTY or EARS)
To print all the short documents on EARS, you can just say
@COPY <ALTODOCS>*.TTY LPT:DEFONT.EP cr

The "notebook" documentation for all the subsystems and subroutine packages may be obtained by:
@COPY <ALTODOCS>SUBSYSTEMS.EARS to LPT: [confirm]
@COPY <ALTODOCS>PACKAGES.EARS to LPT: [confirm]

When you have a change to make to documentation, or wish to introduce new documentation into the system, the following three steps are required:

1. Retrieve the relevant .PUB file from <ALTOSOURCE>. The file name is in the format, sys.PUB, where 'sys' is the name of the subsystem of subroutine package. If you are creating brand new documentation, see the file <ALTOSOURCE>TEMPLATE.PUB for an example.

2. Edit the pub file. If you wish to "try it out," simply pass it to PUB-- a .TTY version of the documentation will be produced.

3. When you are finished, send a message to Barbara Hunt telling her where (on Maxc) she can find the updated pub file. You will probably include this information in the message which you send regarding updated software. Barbara will copy the pub file back to <ALTOSOURCE>, make a .TTY version for <ALTODOCS>, if relevant, and periodically produce new versions of the notebook documentation.

Please be sure to copy the pub files from <ALTOSOURCE> afresh each time you edit them, because they may have been edited to produce expurgated versions (for distribution outside PARC), to produce indexes, remedy formatting problems, etc.

Please try to avoid needless references to PARC or Maxc facilities. For example, it is frowned upon to mention the <ALTO> directory as a place to find something. That is assumed for PARC users. Similarly, avoid needless references to GEARS or EARS.

7.4. New Alto OS - 3/11/74

The new system has the following improvements:

1. The keyboard buffer has been expanded to 100 characters.

2. The undefined opcode trap vector is initialized to send control to location 2, which contains a call to Swat.

3. The underline key (upper-case -) now goes in as control-X.

4. Creating a file is about twice as fast as it used to be.

5. The system maintains the time, using Peter Deutsch's timer routines documented in <ALTODOCS>TIME.TTY. Three of these routines are included in the operating system and defined by INITALTOIO:

   TIMER(tv) - reads the 32-bit millisecond timer into tv!0 and tv!1. Returns tv!1 as its value

   DAYTIME(dv) - reads the current time-of-day (32-bits, with a grain of 1 second) into dv!0 and dv!1. Returns dv as its value.

   SETDAYTIME(dv) - sets the current time-of-day from dv!0 and dv!1. Normally it should not be necessary to do this.

   Peter's package <ALTO>TIME.DM includes CTIME, a Bcpl program which converts the time-of-day into more useful formats for display. It is now a self-contained program, which depends only on procedures declared in INITALTOIO.

6. The time of creation, last write and last read are now maintained in the first six words of the leader block of a file, in that order. They are obtained from the procedures just described. Note that if a file is opened read/write (the default) both read and write times are updated. Therefore, try to specify what you want when you do the open.

7. The leader page for a file can be obtained with ReadFileStuff(stream, v), which reads the leader page into the vector v (which had better have room for 256 words). Currently the only interesting information in the leader page is the time stamps.

8. CALLSUBSYS and OVERWRITE now take a second argument which causes a call of SWAT just before control is given to the subsystem being called if it is true.

9. Using ! as a switch on the subsystem name will cause a call of SWAT just before control is given to the subsystem.

10. Typing <tab> to the command processor will work just like ? (i.e. will make it give you a list of the files which can gotten from what you have typed), except that it will then delete the string it looked up. This is very convenient for interrogating the directory.

7.5. Miscellaneous

The Operating System was designed by Butler Lampson and initially implemented by Gene McDaniel. It is currently being maintained and extended by Butler Lampson.

A program exists for converting CU-format fonts into BR files which have the format given in Table 4.8; see Diana Merry for details.

The files ERRORMESSAGES, OSSYMS, and OSSTRUCTURES referred to in the manual may all be found on the Maxc <ALTO> directory.