ALTO SOFTWARE PACKAGES

Compiled on: June 30, 1975

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

This list is a directory of major Alto software packages. Source files for these programs are available on the <ALTO> directory. The documentation for these packages is saved indivually at .TTY files on <ALTODOCS>. This document is filed as <ALTODOCS>PACKAGES.EARS.

ALLOC: A boundary-tag storage allocator. (Ed McCreight)

BARGRAPH: displays histograms of arbitrary data on an Alto and prints them on Ears. (David Boggs)

BFS: The "basic file system" subroutines. These do page-oriented I/O to disk files organized according to standard Alto conventions. (Butler Lampson)

DPDIVIDE: Computes the quotient and remainder from the division of one 32-bit 2's complement number by another. (Peter Deutsch)

FLOAT: Floating-point package for the Alto that uses no special microcode. (Bob Sproull)

FORMAT: Routines for doing formatted I/O. (Ed McCreight)

GP: General-purpose routines for parsing command lines and the like. (Butler Lampson)

MDI: Subroutine that looks up multiple files in one pass through the directory. (Peter Deutsch)

PROC: Creates BCPL processes for the Alto. (Bob Metcalfe)

PICO: Subroutine package for creating filled-area pictures on the Alto. Also drives Ben Laws' run-code display. Documentation is not on-line; consult a member of the graphics group. (Graphics Group)

RACK: A BCPL storage allocator. (Larry Tesler)

READMU: Subroutine for reading microcode files created by MU. (Chuck Thacker)

ReadPackedRAM: Allows Alto programs which use the RAM to check the constant memory and load the RAM as a part of their initialization. (Peter Deutsch)

SCV: Scan-converts objects from a description of the boundaries of the object. (Bob Sproull)

SPRINT: Subroutines for driving the Diablo printer. (Chuck Thacker)

TIME: Subroutines for converting time-of-day readings to and from human-readable form. (Peter Deutsch)

TRACE: Routines for tracing BCPL procedures. (Peter Deutsch)

VMEM: A software virtual memory package for the Alto. (Peter Deutsch)

ALLOC -- A Basic Storage Allocator

The file ALLOC (.SR for BCPL source, .BR for relocatable binary) 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 allocator maintains as local state the zone from which it is currently allocating. To change or initialize this the user calls NEWZONE. The function ALLOCATE returns a pointer to a block allocated from the current zone. Calling FREE returns a previously-allocated block to the currently allocating zone.

The function INITIALIZEZONE(ZONE, LENGTH, OOPSRTN, CALLERR) initializes the block of storage beginning at ZONE and containing LENGTH words to be a free storage zone. OOPSRTN is taken to be the default allocation error handling routine for the zone. @CALLERR is set to 2 if the block is too small to contain a zone header. INITIALIZEZONE returns ZONE as its value. Thus CALLERR can be omitted, but only if the block being initialized is certainly larger than the zone overhead.

The function NEWZONE(ZONE) sets the allocator's local variable CURZONE to ZONE, and returns the former value of CURZONE.

The function ADDTOZONE(BLOCK, LENGTH) adds the block of storage beginning at BLOCK and containing LENGTH words to the zone pointed to by CURZONE. Its value is a pointer to the zone.

The function ALLOCATE(LENGTH, CALLERR) allocates a block of LENGTH words from the zone pointed to by CURZONE and returns a pointer to that block. CALLERR is an optional parameter. If the allocation cannot be done and CALLERR is present, @CALLERR is set to 1 and the value FALSE is returned to the caller. If the allocation cannot be done and CALLERR is not present, the value returned to the caller is OOPSRTN(CURZONE, LENGTH), where OOPSRTN is the default allocation error handling routine declared by the call on INITIALIZEZONE for the zone pointed to by CURZONE.

The procedure FREE(BLOCK) gives a previously-allocated block of storage back to the zone pointed to by CURZONE. BLOCK must have been the value of a call on ALLOCATE.

The following implementation of the factorial function illustrates the use of ALLOC:

```
static [ Spare
         SpareIsAvail
         ]

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

         let ErrVar = 0
         INITIALIZEZONE(FactZone, 256, StkOvfl, lv ErrVar)
         if ErrVar ne 0 then StkOvfl()

         let OldZone = NEWZONE(FactZone)
         let FactVal = InnerFact(n)
         NEWZONE(OldZone)
```

```
          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(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(stkent)
                    ]

          resultis value
          ]

and StkOvfl(Zone, Length) = valof
          [ unless SpareIsAvail do
                    [ WS("Aargh! Stack stuck!")
                    finish
                    ]
          ADDTOZONE(Spare, 37)
          SpareIsAvail = false
          resultis ALLOCATE(Length)
          ]
```

BARGRAPH -- A Histogram Plotting Package

Bargraph is a package of routines for displaying historgrams of arbitrary data on an Alto display and printing same on Ears. Three other files must be loaded along with Bargraph.br: Format.br, Ctime.br, and Logscale.br. The first two are standard utility packages available on <alto>. Logscale is a bcpl assembly file for logarithmic scaling. Bargraph.br and Logscale.br are available as <alto>Bargraph.dm, a dump file containing the two .br files. The sources are available on <altosource>.

The histograms are 256 bars of maximum height 256 points. Two scaling modes are presently available: Linear and Logarithmic; if someone needs (and comes up with the routine to do) another form of scaling, it is easily added. Bargraph will scale the incoming data so that the largest element is in the upper half of the histogram if the scaling mode is linear.

A bargraph requires 336 scanlines, thus two can be displayed at once. At present, the Alto OS has only two display streams, so before creating the second bargraph, you must give the OS some storage to use for an additional display stream. You do this in the following way:

```
let ds = vec 100
ADDOBJ(5,ds)
```

ADDOBJ is an OS procedure and must be declared external. This is part of the 'undocumented way' of increasing the number of objects in the OS, and is likely to change.

The bargraph display consists of 16 border scan lines at the top, 256 scan lines of graph, 16 border scan lines at the bottom (4 of which are part of the horizontal axis) and 3 lines of text. The first text line displays vertical scaling information and the other two are available to the user as title text. A Bargraph consists of 5 items:

ht	a 256 word vector. ht!i contains the height of bar i (left to right).
cb	a 16 word vector. cb contains the information about a particular instance of Bargraph. This is like a stream handle, and must be passed in all calls to the other procedures in the bargraph package.
db	a 38x336 word vector. This is the bit map area for the display.
ts1	a bcpl string. This is displayed as a line of text below the vertical scaling information at the bottom of the graph. This should be zero if no title string is desired. Do not include carriage returns.
ts2	another bcpl string, displayed below ts1. This should be zero if no string is desired.

CreateBarGraph(db,ht,ts1,ts2,cb)

Draws the vertical and horizontal axes with ticks every 16 dots vertically and every 8 bars horizontally. Displays ts1 and ts2 below the empty graph. Sets up the cb vector. Stores away in cb the creation time (from the OS) of the graph.

UpdateBarGraph(cb,mode,extra)

Recreates the bargraph using the values in ht at the time of the call. cb is the 16 word vector described above. Mode is the scaling mode: 0 = linear, 1 = logarithmic. Extra is a number added to the internal scaling factor before displaying the vertical scaling text. This is a kludgy way to display values greater than 2↑16: when an element of the ht vector overflows, right shift all elements of the vector by 1 and add one to 'extra'. The verti^al scaling information will now say 'max = 2↑17' for linear and '2↑1 to 2↑17' for logarithmic.

PrintBarGraph(cb,mode,s,port)

Generates a Gears format text file which is a fairly faithful copy of what you saw on the display. cb and mode are the same as for UpdateBarGraph. s is a stream opened DISKWOCH. port is a boolean: true for portrait mode and false for landscape mode printing. At the top of the page will be the creation time of the graph (see CreateBarGraph) and the time it was output to the file for printing. Multiple calls on PrintBarGraph with the same stream will append one graph (one printed page) to the file for each call. To print, say 'GEARS/D FOO' where FOO is the name of the disk file.

DestroyBarGraph(cb)

Destroys the bargraph in an orderly way.

If you have more than two sets of data you would like to display (or print), you can call CreateBarGraph for each dataset (!) to set up the graphhandle (cb vector), but use the same db vector (or one of the same two). Since the information for Updateing and Printing is all contained in cb, once it has been set up, Print and Update will do the right thing printing or updating the data in the ht vector passed in the call to CreateBarGraph along with that cb vector (the updated display area will be the db vector passed along with cb to CreateBarGraph).

BFS: Basic File System

The Alto basic file system does page-oriented input/output to disk files organized according to the standard Alto conventions; i. e. each page is identified in its label, and the pages are chained on a doubly-linked list. It also includes facilities for doing disk operations directly, ignoring the file structure. When properly used, it can do input/output at full disk speed.

The basic file system consists of two Bcpl source files, BFS.C and DVEC.C, and one assembly-language file BFSML.C. All these can be found in BFS.DM, together with a file BFSEXT.C which contains some optional procedures.

1. Data Structures

The following data structures are part of the interface between the user and the file system:

pageNumber: pages in a file are numbered from 0. Page 0 is the leader page, and the first data page is page 1. The page number is represented by an integer.

DAs: a vector indexed by page number in which the ith entry contains the disk address of page i of the file, or one of two special values (which can also be found in statics with the same names):
   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.

fileId: a three-word vector which contains the serial number and version number which identify the file. The order of the entries in this vector is the order in which they appear in a disk label, which is unfortunately not the same as the order in which they appear in a directory entry:

```
    fileId:
            version number
            first word of serial number
            second word of serial number
    directory entry:
            first word of serial number
            second word of serial number
            version number
            unused word
            disk address of leader page
```

WARNING: the disk address in the directory entry is a virtual address, but all the basic file system routines expect a real address. The routine RealDA described below will do the conversion.

action: a magic number which specifies what the disk should do. Possible values are:

DCread: check the header and label, and read the data;
DCwrite: check the header and label, and write the data;
DCdoNothing:
DCseekOnly: just seek to the specified track
DCwriteLabel: check the header, write the label and data.

The numbers for these action can be found in statics which are declared external by the basic file system. There are other operations which are possible, but they are not normally available to users.

2. Subroutines

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

ActOnPages(CAs, DAs, fileId, firstPage, lastPage, action, lvNumChars, lastAction, fixedCA, cleanupRoutine). 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 into 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 with the command block (see below) as a parameter after the successful completion of each disk command.

Disk errors are retried five times and then, in the current implementation, call Swat.

Note that the label is not rewritten by DCwrite, so that the number of characters per page will not change. If you need to change the label, you should use WritePages unless you know what you are doing.

WritePages(CAs, DAs, fileId, firstPage, lastPage, lastAction, lvNumChars, lastNumChars, fixedCA). 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 ActOnPages). It fills in pages in the same way as ActOnPages, 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 #1000). It is not necessary for DAs!firstPage to contain a disk address; it may be fillInDA, and a new page will be allocated.

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 DCwrite. It defaults to DCwrite.

In addition to these two routines, there are two others which are not strictly part of the basic file system, but which do not use any other data structures:

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

DeletePages(CA, firstDA, fileId, firstPage) 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.

These two routines can be found on BFSEXT.C, a Bcpl file which can be appended to BFS.C if you want a version of the BFS which has them.

3. Lower Level Use

It is also possible to use the file system at a lower level. This level uses two data structures, zones (defined by the structure CBZ) and control blocks (cb's, defined by the structure CB). The general idea is that a zone is set up with disk command blocks in it. A free block is obtained from the zone with GetCb and sent to the disk with DoDiskCommand. When it is sent to the disk, it is also put on the queue which GetCb uses, but GetCb waits until the disk is done with the command before returning it, and also checks for errors.

If you plan to use these routines, I recommend that you read the code for ActOnPages to find out how they are intended to be called.

InitializeCbStorage(zone, length, firstPage, retry) Zone is the address of a block of length words which can be used to store cb's. It takes at least three cb's to run the disk at full speed; CBzoneLength is a constant which gives the size of a zone which can hold three cb's. FirstPage is used to initialize the currentPage field of the zone. Retry is used to initialzize the retry fields of all the cb's.

GetCb(zone) returns the next cb for the zone. If the next cb is still on the disk command queue, the routine waits until the disk has finished with it. Before returning a cb, GetCb checks for errors. If it finds one, it increments zone>>CBZ.errorCount. If this number is ge the static maxEC, GetCb calls Swat. Otherwise, after doing a restore on the disk if errorCount ge restoreEC, it reinitializes the zone with firstPage equal to the page with the error, and returns to cb>>CB.retry instead of returning normally. The idea is that the code there will retry all the incomplete commands. If there is no error, cb>>CB.cleanupRoutine is called (if it is non-zero) with cb as its argument. Then the numChars field of the label is copied into the currentNumChars field of the zone, and the errorCount field of the zone is cleared. Next, unless GetCb was supplied with a true second argument, the cb is zeroed. Finally, the cb is returned as the value of GetCb.

DoDiskCommand(cb, CA, DA, fileId, pageNumber, action) Constructs a disk command in cb with data address CA, disk address DA, serial and version number taken from fileId, page number taken from pageNumber, and disk command specified by action. It 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
normalWakeups	cb>>CB.zone>>CBZ.normalWakeups
errorWakeups	cb>>CB.zone>>CBZ.errorWakeups

If DA eq fillInDA, the DA field is not set; presumably it is the target of the label for a previous command. Actions are checked for legality (left byte eq #321).

4. Allocating Disk Space

The basic file system also contains routines for allocating disk space. They need to have the following statics supplied to them (values in parentheses are for a Diablo 31):

nTracks:	the number of tracks on the disk (203)
nHeads:	the number of heads per track (2)
nSectors:	the number of sectors per revolution (12)
diskBitTable:	the address of a vector containing the disk bit table
diskBTsize:	the size (words) of the disk bit table (305)

The bit table can be found on the disk in a file called "SYS.STAT". Its format is:

diskBTsize words:	the bit table
1 word:	0
2 words:	the largest serial number used so far

There are four routines:

VirtualDA(real disk address) returns the virtual disk address.

RealDA(virtual disk address) returns the real disk address.

AssignDiskPage(realDA) returns the real disk address of the first free page following realDA, according to the bit table, and sets the corresponding bit. It does not do any checking that the page is actually free (but WritePages does). If there are no free pages it calls Swat.

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

These routines are used by WritePages, CreateFile and DeletePages.

32-by-32-bit division routine

There is now an assembly code routine available to compute the quotient and remainder from the division of one 32-bit 2's complement number by another. This is not a trivial operation (see Knuth, vol. 2, pp. 237 ff.). The calling sequence is

```
        flag = DPDIVIDE(numerator, denominator, quotient, remainder)
```

where each of the four arguments is a pointer to a 2-word vector containing a 32-bit number (high-order word first). If overflow would occur, which can happen only when the denominator is zero, DPDIVIDE returns true and does not affect the quotient or remainder vectors. If no overflow occurs, DPDIVIDE returns false and stores the appropriate results in the quotient and remainder vectors. The remainder always has the same sign as the denominator, and its magnitude lies in [0, abs(denominator)); the quotient is positive if the numerator and denominator have the same sign, negative (if not zero) if they have different signs. DPDIVIDE takes about 5 to 10 times as long as an ordinary 32-by-16-bit division: it does NOT use repeated subtraction and shifting.

FLOAT

FLOAT is a floating-point package for the Alto, intended for use with BCPL. (It uses standard Alto microcode -- no special instructions are needed.) There are 32 floating-point accumulators, numbered 0-31. These accumulators may be loaded, stored, operated on, and tested with the operations provided in this package. 'Storing' an accumulator means converting it to a 2-word packed format (described below) and storing the packed form.

In the discussion below, 'ARG' means: if the 16-bit value is less than the number of accumulators (32), then use the contents of the accumulator of that number. Otherwise, the 16-bit value is assumed to be a pointer to a packed floating-point number.

All of the functions listed below that do not have "==>" after them return their first argument as their value.

1. Floating point routines

FLD (acnum,arg)	Load the specified accumulator from source specified by arg. See above for a definition of 'arg'.
FST (acnum, ptr-to-num)	Store the contents of the accumulator into a 2-word packed floating point format. Error if exponent is too large or small to fit into the packed representation.
FTR (acnum) ==> integer	Truncate the floating point number in the accumulator and return the integer value. Error if number in ac cannot fit in an integer representation.
FLDI (acnum,integer)	Load-immediate of an accumulator with the integer contents (signed 2's complement).
FNEG (acnum)	Negate the contents of the accumulator.
FAD (acnum,arg)	Add the number in the accumulator to the number specified by arg and leave the result in the accumulator. See above for a definition of 'arg'.
FSB (acnum,arg)	Subtract the number specified by 'arg' from the number in the accumulator, and leave the result in the accumulator.
FML (acnum,arg) [also FMP]	Multiply the number specified by 'arg' by the number in the accumulator, and leave the result in the ac.
FDV (acnum,arg)	Divide the contents of the accumulator by the number specified by arg, and leave the result in the ac. Error if attempt to divide by zero.

FCM (acnum,arg) ==> integer	Compare the number in the ac with the number specified by 'arg'. Return -1 IF ARG1 < ARG2 0 IF ARG1 = ARG2 1 IF ARG1 > ARG2
FSN (acnum) ==> integer	Return the sign of the floating point number. -1 if sign negative 0 if value is exactly 0 (quick test!) 1 if sign positive and number non-zero
FLDV (acnum,ptr-to-vec)	Read the 4-element vector into the internal representation of a floating point number.
FSTV (acnum,ptr-to-vector)	Write the accumulator into the 4-element vector in internal representation.

2. Double precision fixed point

There are also some functions for dealing with 2-word fixed point numbers. The functions are chosen to be helpful to DDA scan-converters and the like.

FSTDP(ac,ptr-to-num)	Stores the contents of the floating point ac into the specified double-precision number. First word of the number is the integer part, second is fraction. Two's complement. Error if exponent too large.
FLDDP(ac,ptr-to-num)	Loads floating point ac from dp number.
DPAD(a,b) => ip	a and b are both pointers to dp numbers. The dp sum is formed, and stored in a. Result is the integer part of the number.
DPSB(a,b) => ip	Same as DPAD, but subtraction.
DPSHR(a) => ip	Shift a double-precision number right one bit, and return the integer part.

3. Format of a packed floating point number

```
structure FP: [
    sign bit 1      //1 if negative.
    expon bit 8     //excess 128 format (complemented if number <0)
    mantissa1 bit 7 //High order 7 bits of mantissa
    mantissa2 bit 16 //Low order 16 bits of mantissa
  ]
```

Note this format permits packed numbers to be tested for sign, to be compared (by comparing first words first), to be tested for zero (first word zero is sufficient), and (with some care) to be complemented.

4. Errors

If you wish to capture errors, put the address of a BCPL subroutine in the static FPerrprint. The routine will be called with one parameter:

0 Exponent too large -- FTR
1 Exponent too large -- FST
2 Dividing by zero -- FDV
3 Ac number out of range (any routine)
4 Exponent too large -- FSTDP

FORMAT -- An Output Formatting Package

The file FORMAT (.SR for BCPL source, .BR for relocatable binary) contains a set of subroutines which implement a reasonably nice set of output formatting primitives and a reasonably nice protocol for invoking them. A call of the form

FORMAT(S, F, V1, V2, ..., Vn)

will copy the BCPL string F into the BCPL string S, except that items in F delimited by angle brackets (<>) will be interpreted as format specifications. For those, the format specification and the next input variable Vi will determine what will be put into S. The current format specifications are:

<S> The variable is a BCPL string and is to be copied into S.
<UPS>The variable is an unpacked string (V!0 is the number of characters and V!1 through V!(V!0) are the characters) to be copied into S.
<C> The variable contains a single ASCII character, right-justified.
<D> The variable is numeric, and should be represented as signed decimal.
<UD>..............unsigned decimal.
<B> ....unsigned octal.
<OCT>..............unsigned octal.
<SB> ....signed octal.
<SOCT>..............signed octal.
<BIN> ....unsigned binary.

In addition, the format specifiers take two optional numeric parameters (numbers represented using BCPL conventions) which give the minimum length and fill character to be used in the conversion. For example, <OCT #20 $0> will produce an octal number at least 16 (and, in fact, at most 16) characters long, right-justified and padded to the left with zeros.

FORMATN is exactly like FORMAT except that by a small subterfuge it supplies its own local string, whose address it returns. This string will not change from one call of FORMATN to the next, so that something like WS(FORMATN("It is <D>.", 1975)) will work perfectly.

Finally, the package includes a concatenation routine. After a call of the form

CONCATENATE(D, S1, S2, ..., Sn)

D will be a BCPL string which is the concatenation of the BCPL strings S1, S2, ..., Sn, in that order.

GP: Routines for parsing command lines

The routines described here are a convenient package for parsing command lines and doing a few related functions. They may be found in GP.C (source) and GP.BR (binary). The source needs OSSYMS to compile. No external routines are called except those supplied by the operating system.

An "unpacked string" is a vector *v* such that *v*!1, *v*!2, ..., *v*!(*v*!0) contain the characters of the string, one per word, right justified.

A "parameter" in a command line is a maximal sequence of characters not containing $*S or $*N. All the characters before the first $/ are the "body"; the remaining characters, with any $/ characters ignored, are the "switches". Thus

```
BCPL/F FOO.SR
```

contains two parameters. The first has body "BCPL" and switches "F". The second has body "FOO.SR" and no switches.

SetupReadParam (stringVec, switchVec, stream, comSwitchVec)

- *stringVec* is a vector whose length in words should be greater than the number of characters in the longest body in the command line. A 0 defaults it to a 256-word vector inaccessible to the user; this may be useful if all the parameters of the command are files or numbers (see the discussion of ReadParam below).
- *switchVec* is a vector whose length in words should be greater than the largest number of switches on any unit in the command line. A 0 defaults it to a 128-word vector inaccessible to the user.
- *stream* is an OS *character* stream from which the command line will be read. It will not be RESET or CLOSED. A 0 defaults it to the disk file "COM.CM". The stream is left in the external static *ReadParamStream.*
- *comSwitchVec* is a vector whose length in words should be greater than the number of switches on the first unit in the command line. A 0 defaults it to *switchVec.*

Missing parameters are defaulted.

This routine initializes the parameter-reading machinery. It then does a ReadParam() which will pick off the first parameter (i.e., the name of the program) and leave the name and switches as unpacked strings in *stringVec* and *comSwitchVec.* If either of these was defaulted to an inaccessible vector, the corresponding information is lost.

ReadParam (type, prompt, resultVec, switchVec, returnOnNull)

- *type* is an integer or Bcpl string representing the expected type of the parameter. If *type* < 256, it is interpreted as a character which must select a defined type from the list described below. If *type* > 256 it is treated as a Bcpl string. If the string is one character long, it is interpreted as though that character had been used. If it is longer, the first two characters must select a defined type from the list below.
- *prompt* is a Bcpl string which is used to prompt the user for another try at the parameter if a syntax error is discovered. A 0 defaults it to "Try again: ".
- *resultVec* is a vector used to return the result for types which need more than one word to represent their result. A 0 defaults it to the *stringVec* passed to SetupReadParam.
- *switchVec* is a vector used to return the switches as an unpacked string. A 0 defaults it to the *switchVec* passed to SetupReadParam.
- *returnOnNull* is a boolean which decides what to do if the parameter body is null. It defaults to false.

Missing parameters are defaulted. If *type* is missing, it is defaulted to 0.

One parameter is read from the *stream* passed to SetupReadParam. The switches are separated off and left in *switchVec*. Any $/ characters among the switches are stripped off. If there are no switches, *switchVec*!0 will be 0.

Then the body is handled in a way which depends on the *type*:

0: It is returned in *resultVec* as an unpacked string. Result is *resultVec*.

P: It is returned in *resultVec* as a packed (Bcpl) string. Result is *resultVec*.

I or IC: It is treated as the name of an input character file, to be opened with OPENAFILE(body, DISKROCH). If the open fails, prompt for another name. Result is the stream returned by OPENAFILE. In addition, the file name is returned in *resultvec* as a Bcpl string.

IW: Like I, but a word stream is created.

O or OC: Like I, but GETAFILE(body, DISKWOCH) is called.

OW: Like O, but a word stream is created.

F: Like I, but GETAFILE(body, DISKRW) is called.

EF: Like I, but OPENAFILE(body, DISKRW) is called.

B: An octal number is collected and returned. Numbers may start with #, which forces them octal, and may end with B, b, O, or o (which forces them octal) or with D or d, which forces them decimal. Anything else is a syntax error and causes a prompt for another number. Result is the number.

D: Like B, but for decimal number.

Any undefined type results in a call on Swat.

If the body is empty, ReadParam immediately prompts, without generating an error message from the null body, unless *returnOnNull* is true or *prompt* eq -1, in which case it returns -1 when it sees a null body. When prompting for new input, DEL cancels whatever has been typed and allows another try, and BS and control-A backspace one character.

EvalParam (body, type, prompt, resultVec)

- *body* is an unpacked string
- the other arguments are like the corresponding ones for ReadParam. *resultVec* defaults to *body*.

body and *type* may not be omitted.

Works exactly like ReadParam, using *body* as the parameter body. Does nothing about switches. This routine is useful for programs whose interpretation of parameters depends on the switches attached to them.

ReadString (result, breaks, inStream, editFlag, prompt)

- *result* is a vector in which the string read will be returned, unpacked. May not be defaulted.
- *breaks* is a Bcpl string containing the characters which will cause reading to terminate. Defaults to "*N".
- *inStream* is the stream to read from. Defaults to KEYS.
- *editFlag* says whether DEL, BS and control-A should be interpreted as editing characters. If it is false, they are not. Otherwise they are, and furthermore, *editFlag* is taken as the stream on which echoing of the input should be done. It defaults to false unless *inStream* is KEYS, in which case it defaults to DSP.
- *prompt* is echoed after a DEL. It defaults to "".

Reads characters from *inStream* until one of the characters in *breaks* is encountered, leaving the characters read in *result* as an unpacked string. Returns the break character. Allows editing of the input as described under *editFlag* above.

DefaultArgs (lvNa, first, d0, d1, ...)

This routine should be called only in the following context:

```
and    Foo (a0, a1, a2, a3; numargs na) be [
           ...
           Default Args (lv na, 1, "alpha", 12)
```

- *lvNa* is the lv of the numargs formal (na in the example), which MUST have been present in the declaration of the routine or function which calls DefaultArgs. It must not be omitted.
- *first* is the number of the first argument which may be defaulted, counting from 0. It defaults to 0. If fewer than *first* arguments were supplied to the calling routine, Swat is called. If *first* is negative, its absolute value is used, and actual arguments from *first* on which are zero are replaced by the corresponding default values.
- *d0, d1, etc.* are the default values for arguments p!*first*, p!(*first* + 1), etc. There must not be more than 10 of them.

Checks that at least *first* arguments were supplied to the caller, and calls Swat if not. Let ai be the last argument supplied to the caller. Sets a(i + 1) = *d*(i - *first*), a(i + 2) = *d*(i - *first* + 1), and so on for all the parameters in the caller's formal parameter list. If not enough *d*s were supplied, the last one is used repeatedly.

In the example above, a call of Foo(n) will result in

```
a0 = n
 a1 = "alpha"
 a2 = 12
 a3 = 12
```

after the call of DefaultArgs.

AddItem (vek, value)

- *vek* is a vector whose current size is given by *vek*!0.
- *value* is an uninterpreted 16-bit quantity.

Increments *vek*!0 and stores *value* at the new *vek*!(*vek*!0).

MDI: Multiple Directory Lookups

There is now available a routine to look up a group of file names in a directory in a single pass, and return the directory entries without actually opening the files. This may be useful for programs (such as BLDR) which wish to avoid time-consuming multiple scans of a directory. It may be found on MDI.BR.

The code is written in BCPL. It declares one entry procedure LOOKUPENTRIES, and only uses standard procedures from the operating system.

LOOKUPENTRIES(S, NAMEVEC, PRVEC, CNT, FILESONLY)

S is a directory: it must be a disk stream (for example, SYSTEMDIR). LOOKUPENTRIES resets S and then reads through it. NAMEVEC is a vector of CNT strings, the file names. PRVEC is a vector of DIRPREAMBLESIZE*CNT words, where LOOKUPENTRIES stores the directory preambles corresponding to NAMEVEC. If a given name is not found, its block in PRVEC will be zeros: since the first word of a directory entry can never be zero, one can test the first word of the PRVEC block to determine if a name was found. If FILESONLY is true, LOOKUPENTRIES will only check directory entries that designate real files; if false, LOOKUPENTRIES will check all entries (including links, or any other types that may be defined eventually).

LOOKUPENTRIES returns the number of names not found. Thus if all names were found, LOOKUPENTRIES returns zero.

Process Primitives

These Process Primitives are a minimal set of procedures for creating BCPL processes for the Alto.

The example program BOUNCE.C (2 pages of BCPL with PROCC and PROCA) is intended to teach the use of these procedures; you must study the example while reading this documentation. An understanding of the Alto interrupt system is helpful.

The process procedures establish and maintain a set of contexts, each normally associated with an Alto priority interrupt level. The levels are numbered from 1 to 16 and carry enough state so that a BCPL-coded process can operate in each. Level 1 has the highest priority and is currently assigned to a parity process by the operating system; level 9 is used by SWAT; and level 13 is the keyboard process.

A program wishing to use the primitives must first call InitProcessSystem() which (1) clobbers BCPL's GETFRAME so that it will fall into SWAT if a stack overflow occurs, (2) establishes the main program level (16) as a process with a PCB (process control block), and (3) fixes BCPL's FINISH to restore the state of the interrupt system before returning to the operating system. The 1st and 3rd of these functions will not be required when GETFRAME is fixed and when the OS properly initializes the interrupt system between subsystems. The first function is important, you should note, because there can now be more than one stack to overflow.

To create a process, BeginProcessInit(Level,Stack,Size) and EndProcessInit(StartPC) must be called. The process must be assigned an interrupt level (Level=1-15) and some space for a BCPL stack if required (say Stack=GetFixed(100) and Size=100). BeginProcessInit returns the address to which the Alto should transfer control when the process is to be awakened. EndProcessInit finishes initialization of the process and provides the process's starting PC. The initial context of the process is a copy of the stack frame of the procedure calling BeginProcessInit. Between the call to BeginProcessInit and the matching call to EndProcessInit, the calling procedure executes in the context of the new process and initialization can be performed. After initialization, the process is ready to be awakened. Normally, the process's wakeup address is placed in the Alto's interrupt vector, the associated level is enabled, and wakeups are begun. Wakeups can be generated either by some input-output device (e.g., the display, the Ethernet) or by another process. (See BOUNCE.C)

The procedure WakeUpBits(Bits) ORs Bits into the Alto interrupt system's WakeUpsWaiting word thus causing interrupts to take on the specified interrupt channel(s) if activated.

The procedure WakeUp(Level) checks that Level is a legal level number (1-16) and that the interrupt vector is fixed up to receive an interrupt on that level. If so, it computes the appropriate interrupt bit and calls WakeUpBits.

The procedure Block() is executed by a process to stop computing and await a new wakeup. Block() saves the state of the calling process and restores the state of the preempted lower-priority process.

Once awakened, a process continues to run until either it blocks or is preempted by a higher priority (lower numbered) process. Thus, a process is in one of three states: (1) running, (2) suspended, or (3) blocked.

The procedures DisableInterrupts() and EnableInterrupts() do an Alto DIR and EIR, respectively.

The procedure MoveBlock(Dst,Src,Num) moves Num memory words from Src to Dst.

The sources PROCC.C (4 pages of BCPL) and PROCA.A (4 pages of ALTOASM) are in <ALTOSOURCE>PROCSOURCES.DM; the BRs are in <ALTO>PROCBRS.DM.

We plan to add these process primtives to a future version of the Alto OS. Earlier versions of these procedures are now being used in Ears, Gears, Maxc, FTP, Bravo, and assorted others.

A companion set of utility procedures for process scheduling and communication exists and is now being documented.

Two improvements to these procedures are already on a list. The first is to generalize the stack frame allocation scheme now in GETFRAME so that frames can be allocated from a heap. The second is to provide procedures for switching among numerous processes at level 16. Can you suggest some others?

```
//bounce.c - Example use of process primitives - Bob Metcalfe
//bldr bounce procc proca initaltoio

external
        [
        INITALTOIO; GetFixed; WS; WO
        BeginProcessInit; EndProcessInit; InitProcessSystem
        DisableInterrupts; EnableInterrupts; Block; WakeUp
        ]

manifest
        [
        AdrCursorMap=#431
        AdrCursorX=#426; AdrCursorY=#427 //Cursor X&Y
        AdrMouseX=#424; AdrMouseY=#425 //Mouse X&Y
        MouseLevel=15; FlashLevel=14 //Process levels
        ]

//Main program; initializes processes and loops WSing
let BounceMain() be
[BounceMain
INITALTOIO() //Link to OS IO
InitProcessSystem()
@#453=#401 //Swat and parity wakeups only (not old keyboard)
InitFlashCursor(FlashLevel,GetFixed(50),50) //Make cursor flash
InitBounce(MouseLevel,GetFixed(50),50)
                                //Strange mouse control of cursor
[ WS("*NCursor XY "); WO(@AdrCursorX); WO(@AdrCursorY)] repeat
]BounceMain

//Process which makes cursor a bouncing ball with shove from mouse
and InitBounce(Level,Stack,Size) be
[InitBounce
let WakeBit=1 lshift (Level-1) //Bit of chosen process level
@(#500+Level)=BeginProcessInit(Level,Stack,Size) //Int dispatch
@AdrCursorX=0; @AdrCursorY=0 //At top left
@AdrMouseX=0; @AdrMouseY=0 //No shove
let CursorSpeedX=0; let CursorSpeedY=0 //Not moving
EndProcessInit(MouseFix) //Process init ended; give init PC
DisableInterrupts()
@#421=(@#421)%WakeBit    //Arrange periodic wakeup from display
@#453=(@#453)%WakeBit    //Activate interrupt
EnableInterrupts()
return //Process initialized and running, proceed at level 16

MouseFix: //Comes here when process first awakened
CursorSpeedX=Limit(CursorSpeedX+(@AdrMouseX/2),-30,30) //Shove cursor
CursorSpeedY=Limit(CursorSpeedY+(@AdrMouseY/2),-30,30)
@AdrCursorX=Limit(@AdrCursorX+CursorSpeedX,0,606-16) //Cursor is moving
@AdrCursorY=Limit(@AdrCursorY+CursorSpeedY,0,808-16)
@AdrMouseX=0; @AdrMouseY=0 //Incremental shoves
if (@AdrCursorX le 0) then CursorSpeedX=Abs(CursorSpeedX) //Bounce
if (@AdrCursorY le 0) then CursorSpeedY=Abs(CursorSpeedY)
if (@AdrCursorX ge 606-16) then CursorSpeedX=-Abs(CursorSpeedX)
if (@AdrCursorY ge 808-16) then CursorSpeedY=-Abs(CursorSpeedY)
if ((@AdrCursorX eq 0)&(@AdrCursorY eq 0)) then
        [
        @AdrMouseX=0; @AdrMouseY=0 //No shove
        CursorSpeedX=0; CursorSpeedY=0 //Not moving
        ]
WakeUp(FlashLevel) //Provide periodic wakeup of flashing
Block()
//Done for now, let lower processes run until next display wakeup
goto MouseFix //Awake again, do the same thing again
]InitBounce

//Process which flashes cursor (gray and black slowly)
and InitFlashCursor(Level,Stack,Size) be
[InitFlashCursor
let WakeBit=1 lshift (Level-1) //Bit of chosen process level
@(#500+Level)=BeginProcessInit(Level,Stack,Size) //Int dispatch
EndProcessInit(BeginFlash) //Process init ended, give init PC
DisableInterrupts()
@#453=(@#453)%WakeBit    //Activate interrupt
EnableInterrupts()
return //Process initialized and running, proceed at level 16

//Periodically sweep cursor on and off (driven by mouse routine)
BeginFlash:
for i=0 to 15 do [ @(AdrCursorMap+i)=-1; Block()] //Gradually blacken
for i=0 to 15 do [ @(AdrCursorMap+i)=#125252; Block()] //Now gray
goto BeginFlash
]InitFlashCursor

and Abs(Value)=valof
[ test (Value ls 0) ifso resultis (-Value) ifnot resultis (+Value)]

and Limit(Val,Min,Max)=valof
[ if (Val gr Max) then Val=Max; if (Val ls Min) then Val=Min;
 resultis Val]
```

"RACK" Storage Allocator

May 7, 1975 (rev. May 14, 1975)

The RACK is an Alto BCPL storage allocator with the following properties:

When a record is created, both a movable "instance" and an immovable "finger" are allocated, each pointing at the other.

Every record is accessed indirectly through its finger.

Instances are allocated in a stack-like manner.

If the rack overflows trying to allocate, it compacts.

Incremental compaction can be done during waits.

Records that exist for a long period of time sift to the bottom of the rack and do not take the time of the compacter.

Every record belongs to a "class". Most classes have fixed-length instances, but class "array" has variable length instances.

Every instance has a three word header. Counting the finger, the total overhead is four words per record. The header contains: Back pointer to the finger; Class; Length of Data.

A record can be "changed" by repointing its finger; for example, the length of an array can change;

There may be several racks, each with its own fingers.

This document describes the highlights of the package. For more details, see the comments before each procedure in the source code.

1. VECTORS

It is assumed that the caller has a vector allocator to allocate immovable permanent storage in memory. As you know, a vector 'x' is accessed by:

```
    Structured vector access
            x>>S.f.

    Subscripted vector access
            x ! i
```

The RACK package provides the following procedures to operate on vectors:

```
    FillVec(dest, value, nwords)
            dest[0:nwords-1]←value
    ZeroVec(dest, nwords)
            dest[0:nwords-1]←0
    MoveVec(dest, source, nwords)
            dest[0:nwords-1]←source[0:nwords-1]
```

2. RECORDS

A record 'x' is accessed through its finger by:

```
        x>>o     OR      R(x)     OR      W(x)
```

These are all equivalent, but the first form is fastest. They all provide indirect access to the record through a finger. R and W are provided for users who expect later to upgrade their storage management system to one which has more complex access, such as virtual memory. Similarly, the redundant functions Rsub(x,i) and Wsub(x,i,value) are provided for compatibility with more complex access methods, such as virtual memory and hash tables.

```
        Read from structured record
                value = x>>o>>S.f
                value = R(x)>>S.f

        Write into structured record
                x>>o>>S.f = value
                W(x)>>S.f = value

        Read from subscripted record
                value = x>>o ! i
                value = R(x) ! i
                value = Rsub(x, i)

        Write into subscripted record
                x>>o ! i = value
                W(x) ! i = value
                Wsub(x, i, value) // resultis value
```

The procedures in this package take fingers as their arguments and return fingers as their values -- never instances. IT IS STRONGLY RECOMMENDED THAT ALL REFERENCE TO A RECORD BE THROUGH ITS FINGER. Such caution will prevent BUGS due to the instance moving out from under you.

3. MAKING RECORDS

An array "tbl" of n elements can be made by:

```
        let tbl = Array(n)
```

A structure "e" of class "employee" can be made by:

```
        let e = New(employee)
```

All fields of a new record are zero.

4. FREEING

A record is freed by:

```
        Free(record)
```

This frees both the finger and the instance. Further access to the record is impossible. Its space will be reclaimed during a subsequent compaction.

5. FILLING RECORDS

```
Fill(dest, value)
        Every field is set to "value".
        E.G., let tbl = Fill(Array(25), -1)

Zero(dest)
        Every field is set to 0.

Load(dest, a, b, c, ...)
        The record is loaded word by word from a, b, c, ...
        E.G., let e = Load(New(employee), "Joe Doe", 52, $M)
        Limit: 33 fields.

Move(dest, source)
        The fields of dest are copied from those of source.
        (There are other calling sequences to move subrecords.)

dest = Copy(source)
        Makes a duplicate of source.
```

6. MULTIPLE RACKS

Records are allocated on the rack "defaultrack" unless otherwise specified. The following functions have an optional last argument which overrides defaultrack:

```
        Array(length, rack)
        Copy(source, rack)
        New(class, rack)
```

The rack that contains a certain record can be determined by:

```
        RackWith(record)
```

7. COMPACTING

If you want to compact a little bit of a rack from a user wait loop, call:

```
        CompactRack(rack)
```

If the rack is already compact and nothing needs to be moved, it returns false. Otherwise, it moves one instance and returns true.

8. LENGTHS

The length of data in any record can be determined by:

```
        Length(record)
```

The length of an array record can be changed by:

```
        ChangeLength(tbl, newlength)
```

What this does is to allocate a new array, move the data from the old one, swap the fingers, and deallocate the old instance and the new finger. If you would like to play similar games with other data types, see the procedures ChangeLength() and Change() in RACK.C.

9. CLASSES

The class of a record can be determined by:
```
        Class(rec)
```

To make a new class 'employee', do this:
```
        structure EMPLOYEE:
                [
                name    word    // string
                age     word    // integer
                sex     word    // character
                ]

        employee = MakeClass("employee", size EMPLOYEE/16)
```

To make a class of variable length instances, imitate the code for 'classarray' in RACK.C.

10. LOADING AND INITIALIZING THE PACKAGE

Load RACK.BR and RACKML.BR (2300b words) with your program. The program should include code like this:
```
        get "RACK.DF" // declares external statics such as 'rackvec'

        structure XXX: // or whatever classes you have
                [
                fld word
                ...
                ]
        static  [
                rackyyy // or whatever racks you want
                rackzzz
                ...
                classxxx        // or whatever classes you have
                ...
                ]

        let main(v) be
        [
        ...
        let t = vec maxnumracks // maxnumracks=10 in RACK.DF
        rackvec = t
        rackyyy = MakeRack(<bot>,<top>)
        rackzzz = MakeRack(<bot>,<top>)
        MakeBasicClasses()
        classxxx = MakeClass("xxx", size XXX/16)
        ...
        ]

        and
        let Error(string) be Swat()
```
The first rack created by MakeRack() becomes the initial 'defaultrack'.

11. BASIC CLASSES

MakeBasicClasses() makes classes called "classclass" and "classarray". Instances of class "classclass" have two fields: "title" and "length". For example:

```
        Class(New(employee)) : employee
        R(employee)>>CLASS.title : "employee"
        R(employee)>>CLASS.length : 3
```

Classes of variable length instances do not have a classwide length, so:

```
        R(classarray)>>CLASS.length : 0
```

12. OTHER PROCEDURES

See RACK.C for more information about the procedures mentioned above, plus:

```
        ChangeClass(record, newclass)
        EnumerateRacks(proc)
        EnumerateRecords(proc, rack, justclass)
        FindRecord(proc, rack, justclass)
        InRack(record, rack)
        SwapInstances(record1, record2)
```

In RACKML.A are provided:

MulDiv(a,b,c)	a*b/c, with interim product in double precision
Min(a, b)	Signed minimum
Max(a, b)	Signed maximum
Umin(a, b)	Unsigned minimum
Umax(a, b)	Unsigned maximum
Usc(a, b)	Unsigned compare = a<b?-1, a=b?0, a>b?1
Swat()	

13. POSSIBLE IMPROVEMENTS

Users who are convinced by the SPY or by other methods that RACK performance is inadequate for their needs can improve it in several ways. There are various places where space can be traded off for speed, speed of allocation for speed of compaction, speed for code length, and so forth. Certain routines are good candidates for machine code, notably Class(), Length(), InRack(), and RackWith().

If you have only one rack, you can speed up Free() and certain other functions by replacing every call of "RackWith(...)" by simply "defaultrack". If you have only two racks, RackWith() can be simplified to an unsigned compare.

If you regard all records as arrays, you can eliminate the entire class mechanism and combine the procedures Array() and Allocate() into a single routine.

14. CHECKING PROCEDURES

It is not necessary to load CHECK.C to run the RACK package, but if it is loaded (in 1200b words) it provides:

Error(string)	A procedure of this name is required by RACK.C. This version of Error just calls Swat().
Check(string)	Calls Error(string) unless string=false.
Identity(x)	Returns x
Random(lo, hi)	A random integer between lo and hi inclusive.

There are also routines with names like Badxxx() which are called like this:

Badxxx(x)	Returns false if x good, returns an error string if x bad
Badxxx(x, proc)	Returns false if x good, calls proc(string) if x bad

Badxxx(x, Check) is equivalent to Check(Badxxx(x)) except in the former case Check is called from within Badxxx so Swat is more helpful.

BadName(string [,proc])	Approves strings of letters and digits.
BadRack(rack [,proc])	Approves well-formed racks.
BadRecord(record [,proc])	Approves consistent records.
BadOrder(a1,...,an) and BadStrictOrder(a1,...,an)	Test for unsigned ascending sequence. They may take 1 or more arguments, but no proc.

Checking procedures can be used to close in on mysterious bugs.

Another condition you might want to check is that the number of records in the rack changes by a certain number (like 0) during a computation. The number of records in rack "rackyyy" is determined by:

```
        rackyyy>>RACK.records
```

15. SUPPORT

This package will be maintained by the author (Larry Tesler).

16. NEWS

May 14, 1975: Changed RACK.DF and RACK.C -- Fixed bugs; made Enumerators be routines instead of functions; added FindRecord().

READMU

A library routine is now available for reading MU binary output. This routine may be useful for those interested in debugging, analyzing, or otherwise manipulating Alto microcode. The package is called READMU; it is written in BCPL and the only file required to use it is READMU.BR. It declares one entry procedure, ReadMU, and one entry static, MuSeqNo. The arguments to ReadMU are (stream, writeram, writecon, definename) of which only stream is required. Their significance is as follows:

stream must be a word-oriented input stream, the MU binary file. ReadMU only reads from this stream.

writeram(addr, hipart, lopart) is called for every instruction in the file. If the writeram argument is missing or 0, instructions are discarded.

writecon(addr, value) is called for every constant in the file. If writecon is missing or 0, constants are discarded.

definename(addr, string, memoryid) is called for every symbol definition in the file. memoryid is $R for R registers, $C for constants, or $I for instructions. If definename is missing or 0, symbol definitions are ignored.

MU outputs instructions in an unspecified order, but with each instruction it outputs a "sequence number" that reflects the order of appearance of the instructions in the source file. ReadMU leaves this sequence number in the static MuSeqNo for use by the writeram procedure.

ReadMU returns 0 if everything went normally. If an error occurs, ReadMU returns immediately (leaving the stream positioned just past the item in error) and the value returned is a string which identifies the type of error. ReadMU detects the following errors:

Unexpected end of stream
Bad memory #
Data for undefined memory
Bad width
Bad memory name
Invalid block type

PACKMU, RPRAM, READPRAM

There are now available two subsystems and a library routine which make it easy for Alto programs which use the RAM to check the constant memory and load the RAM as part of their initialization. The first subsystem, PACKMU, takes the output of MU (a .MB file) and converts it to a "packed RAM image" which is easy to load. The second subsystem, RPRAM, reads a packed RAM image, checks the constant memory, and loads the RAM. This function is also available through a library routine.

A packed RAM image is a file containing 4400b words. The first 400b words are the desired contents of the constant memory: a zero word (which MU cannot generate) means "don't care". The remaining 4000b words are the contents of the RAM. Each instruction occupies two words, first high-order part, then low-order part, e.g. words 0 and 1 go into RAM location 0, words 2 and 3 into RAM location 1, and so on.

The invocation format for PACKMU is >PACKMU mbfile pramfile where mbfile is the output from MU (normally something.MB) and pramfile is the file for the packed RAM image. PACKMU prints out xxx constants, yyy instructions to indicate the number of constants and instructions read from mbfile. If mbfile is somehow illegal, PACKMU prints Error: and an error message instead.

The invocation format for RPRAM is >RPRAM pramfile where pramfile is the output from PACKMU. If there are any disagreements between the constants in pramfile and the actual constant memory, RPRAM prints Constant nnn is xxx, should be yyy for each constant that disagrees, and a summary message nnn constants differ at the end of loading (but it still loads the RAM). If there are no disagreements, RPRAM prints nothing.

The subroutine is called ReadPackedRAM(stream). The argument should be an open, word-oriented input stream positioned at the beginning of a pramfile. ReadPackedRAM does exactly the same thing as the RPRAM subsystem, including printing disagreement messages (on the standard display stream DSP), but instead of printing the summary message it just returns the number of disagreements. RPRAM essentially just opens the pramfile and calls ReadPackedRAM.

Maintainer's notes:

PACKMU uses the library packages GP and READMU.

RPRAM uses the library package GP.

SCV: Scan Converter Package

SCV is a package for scan-converting objects from a description of the boundaries of the object. The package computes which bits of each scan-line fall under the object described; if these bits are displayed in black, the object will appear, colored black.

The input to SCV is an ordered sequence of edge descriptions; an edge may be either a straight line or a spline curve. SCV scales the coordinates of the edge and computes the intersections of the edges with the coordinate grid. Finally, the intersections are sorted, first by scan-line number, and then by "run direction" within the scan-line.

Thus the coordinate system is based on "scan-direction" and "run-direction" rather than on x and y. The coordinates of a point are (s,r) where s is the scan-line number, and r is measured along the scan-line. For example, on the Alto, s might run from 0 to 807, a vertical measure; r might run from 0 to 605, a horizontal measure.

Before passing to detailed explanations, consider the following example:

```
SCVBeginObject(false)            (5,10)
SCVMoveTo(0,0)                    /\
SCVDrawTo(10,0)                  /  \
SCVDrawTo(5,10)                 /    \
SCVEndObject(v)                /      \
 ...(details)                 /--------\
SCVReadRuns(v,buf,100)      (0,0)     (10,0)
```

This returns a list of intersections: (1,0) (1,2) (2,0) (2,4) (3,0) (3,6) (4,0) (4,8) (5,0) (5,10) (6,0) (6,8) (7,0) (7,6) (8,0) (8,4) (9,0) (9,2) (10,0) (10,0). If these intersections are paired into "runs," we can see which bits to turn on (e.g. on scan-line 3, we turn on bits 0 (inclusive) through 6 (exclusive); more on this below). Thus we get (remember, scan-lines are vertical in the above example):

```
    *
    *
   ***
   ***
  *****
  *****
 *******
 *******
*********
*********
```

Initialization

SCVInit(Getb,Putb,Error)

This routine must be called before any objects are scan-converted. Getb is the address of a routine for obtaining blocks of storage; Putb is a routine to return these blocks to the pool; Error is an error routine. Templates for these subroutines are:

```
let
Getb(BlockSize) = valof [
  //Get a free storage block of length BlockSize.
  //Suppose Addr is the address of the first usable word.
  resultis Addr
  ] and
```

```
Putb(Addr) be [
  //Returns block acquired previously by Getb.
  ] and
Error(String) be [
  //String is a BCPL string that describes the error.
  ]
```

SCVMatrix(a,b,c,d)

This routine sets the scaling matrix. In all functions that have s and r values as parameters, the following scaling takes place:

```
S = a*s + c*r
R = b*s + d*r
```

and the values of S and R are actually used. In all explanations below, if upper-case S and R are used, they represent scaled versions of s and r. The arguments to SCVMatrix are either:

a. 0. The corresponding coefficient is zero.

b. A pointer to a packed floating-point number.

c. The number of a floating-point accumulator. (See "Restrictions," below.)

Thus the identity transformation can be established with: FLDI(2,1); SCVMatrix(2,0,0,2).

SCVTransformF(s,r,v)

This routine scales s and r by the scaling matrix, and returns Floor(Round(S)) in v!0 and Floor(Round(R)) in v!1. The full value of S is left in floating-point accumulator 8; that of R in accumulator 9.

Generating Object Descriptions

The operations of generating object descriptions and of actually computing the intersections are separated in order to cater to certain applications. The object generation process is: (1) initialize by calling SCVBeginObject, (2) pass boundary descriptions to SCVMoveTo, SCVDrawTo or SCVDrawCurve, and (3) finish by calling SCVEndObject, which returns an object descriptor (structure SCV).

SCVBeginObject(Care)

Called to begin describing a new object. Care is true if "careful" scan conversion is required (see SCVEndObject).

SCVMoveTo(s,r)   -or-   SCVMoveToF(s,r)

Starts a new boundary, and sets the "current" point to (S,R). The arguments to SCVMoveTo are signed 16-bit integers; SCVMoveToF is identical in function, but requires floating-point numbers (or accumulator numbers) as arguments.

SCVDrawTo(s,r)   -or-   SCVDrawToF(s,r)

Specifies that the next leg of the boundary is an edge from the "current" point to (S,R). The current point is set to (S,R). The arguments to SCVDrawTo are signed 16-bit integers; SCVDrawToF is identical in function, but requires floating-point numbers (or accumulator numbers) as arguments.

SCVDrawCurve(sa,ra,sb,rb,sc,rc)

Specifies that the next leg of the boundary is a parametric cubic spline traced out by values of t from 0 to 1 in the equations ("current" point is (So,Ro)):

```
    S(t) = So + Sa t + Sb t↑2 + Sc t↑3
    R(t) = Ro + Ra t + Rb t↑2 + Rc t↑3
```

The "current" point is set to (S(1),R(1)). Arguments are floating-point numbers (or accumulator numbers).

SCVEndObject(v)

Finishes the object descr'ption, and returns useful data in v:

v>>SCV.Smin, v>>SCV.Smax. Minimum and maximum values of S (inclusive) where the object lies. Signed 16-bit integers.

v>>SCV.Rmin, v>>SCV.Rmax. Minimum and maximum values of R (inclusive). (If splines are used, these two numbers are accurate only if the Care argument to SCVBeginObject is "true".) Signed 16-bit integers.

Generating Intersections

Armed with an object description ("v" argument to SCVEndObject), intersections can be calculated with calls to SCVReadRuns.

SCVReadRuns(v,Buffer,Bufsize)

Calculates some intersections, and records them in a buffer (Buffer is the address of the first usable word of the buffer, Bufsize is the number of words in the buffer). Two values in the vector v govern the range of S values to consider: values from v>>SCV.Sbegin and v>>SCV.Send (inclusive) are considered. NB: This S range must proceed unhesitatingly from v>>SCV.Smin to v>>SCV.Smax, as returned by SCVEndObject.

The function returns, in v:

v>>SCV.IntPtr. Pointer to the first intersection.

v>>SCV.IntCnt. Number of intersections calculated. This is guaranteed to be even, so that an integral number of intersection pairs ("runs") are in the buffer.

v>>SCV.Send. Largest S value considered. If the buffer is too small to contain all intersections in the S range requested, the range is reduced until the intersections will fit. On return, v>>SCV.Sbegin and v>>SCV.Send represent the range actually calculated.

The intersections returned by SCVReadRuns are sorted in the buffer by S and then by R. Each intersection requires two words: the first is the S value, the second the R value.

The following code demonstrates a probable use of SCVReadRuns:

```
SCVBeginObject(false)
...specify boundaries...
let v=vec size SCV/16
SCVEndObject(v)

let b=vec 200
v>>SCV.Sbegin=v>>SCV.Smin        //First range
        [
        v>>SCV.Send=v>>SCV.Smax //Assume entire range fits.
        SCVReadRuns(v,b,200)    //Calculate intersections.
        let n=v>>SCV.IntCnt
        if n eq 0 then break    //All done.
        let p=v>>SCV.IntPtr
        for i=1 to n by 2 do    //Loop for each run.
                [
                let S=p!0               //S value
                for R=p!1 to p!3-1 do TurnOnBit(S,R)
                p=p+4           //Next intersection pair.
                ]
        v>>SCV.Sbegin=v>>SCV.Send+1 //Prepare next S range.
        ] repeat
```

The loop on R values of the intersection pair stops just short of the second intersection. That the R interval should be open can be demonstrated with the following example: suppose that two edges intersect a particular scan-line at R=0.5 and R=2.5. Clearly the "width" of the object on this scan-line is 2.5-0.5=2.0. SCV truncates the R values before sorting them, and so reports intersections at R=0 and R=2, again a "width" of 2.

Operation

SCV code is contained in the files SCVMAIN.C and SCVSORT.C. The definitions for the SCV structure are in SCV.DFS. The SCV package requires the floating-point package FLOAT. The program SCVTEST.C is an example of the use of SCV.

Strategies

The orderly way in which SCVReadRuns proceeds from small values of S to large values can sometimes be linked to the order in which information is used, e.g. added to the screen. If several objects are to be added in one pass over the screen, SCV can handle that as follows:

a. Generate object descriptions for all objects, saving the "v" vectors for each one.

b. Call SCVReadRuns for each object, dumping intersections into separate buffers. Use the intersection information to update the screen. (Or, for the energetic, merge the runs from the several objects!)

c. Repeat step b until all objects are finished.

Note that objects may have several closed boundaries (a call to SCVMoveTo signals the beginning of a new boundary). The most common use of this feature is to specify the boundaries of "holes" in the object.

Restrictions and Caveats

1. After scaling, S and R must both lie between -16000 and +16000.

2. The SCV package uses many floating-point accumulators. However, it guarantees never to clobber AC 0 to 7 inclusive. Similarly, the caller must guarantee:

   a. Not to clobber AC's 28-31 inclusive unless he is willing to re-establish the scaling matrix with a call to SCVMatrix.

   b. Not to clobber AC's 22-27 inclusive during object generation (i.e. between a call to SCVBeginObject and SCVEndObject).

3. If you do not intend to use splines at all, the code in SCVMAIN.C can be shortened considerably. Remove all code between comments //BEGIN $$$ and //END $$$. (Eventually, conditional compilation will be used.)

4. Free storage use. For each edge, an 8 word block is acquired (24 if it is a spline); the blocks are released by SCVReadRuns when it is no longer needed.

Diablo Printer Utility

SPRINT.DM contains an Alto .BR file with 4 routines for driving the Diablo printer:

INITPRINTER(v1,v2) must be called before doing anything else. It takes two vec 255's which it uses throughout the life of your program. It returns true if it could restore and init the printer, else false.

DCHAR(CHAR) prints the character

DSTR(STRING) prints the string

DOCT(NUMBER) prints the octal number, unsigned, with leading 0's converted to spaces.

Things are not printed until an entire line (132 chars or to a CR) is supplied. Tabs are 8 spaces, and a page eject is done every 55 lines. If the printer hangs in some awful way, a message is printed on the system display area, and a "finish" is done.

Daytime and interval timing package

There now exist a pair of packages which provide the following useful facilities for Alto programs:

> The "timer" package, which provides (the illusion of) a continuously running timer with a grain of 1 millisecond and a width of 32 bits, thus a period of about a month, and (the illusion of) a time-of-day clock with a grain of 1 second and a width of 32 bits, origined at 1901 and good through about 2050. These routines are now available in the operating system itself. See the Operating System Reference Manual.
>
> The "daytime" package, which provides for converting time-of-day readings to and from human-readable form.

The chief value of the timer package is that it continues to function properly without losing time even if the Alto is booted, provided that page 1 is not clobbered and that the Alto does not remain non compos mentis for longer than the period of the hardware clock (about 20 minutes). Even in this case, timing will resume properly if one obtains the correct time of day from some other source and informs the timer package thereof; of course, the accuracy of timings spanning such an event is dependent on the accuracy of the new time.

1. Daytime

The daytime package is written in Bcpl. It is found in CTIME.BR. It defines 7 procedures (UNPACKDT, PACKDT, WRITEUDT, CONVUDT, FINDMONTH, MONTHNAME, WEEKDAY). It requires the timer package. The procedures do the following:

> UNPACKDT(dv, uv) - dv!0 and dv!1 contain a time-of-day. (If dv=0, uses the current time from DAYTIME.) Unpacks this into uv!0 through uv!6 as follows:
>
> uv!0 - actual year (e.g. 1974)
> uv!1 - month (January=0)
> uv!2 - day of month (first day=1)
> uv!3 - hour of day (midnight=0)
> uv!4 - minute
> uv!5 - second
> uv!6 - true if daylight saving time in effect
>
> PACKDT(uv, dv, dstflag) - performs the inverse of UNPACKDT. Returns 0 if successful; otherwise, returns 1+j, where uv!j was illegal (e.g. returns 2 if the month was invalid). If dstflag is not supplied or false, assumes uv is the result of converting a string, and uses daylight saving time if appropriate to the date in uv (ignoring uv!6); if dstflag is true, uses uv!6 to decide whether daylight saving time is in effect.
>
> WRITEUDT(strm, uv) - takes an unpacked time-of-day (in uv!0 through uv!6) and writes it on the stream strm in the form 29-DEC-74 18:39:47. If uv=0, uses the current time from DAYTIME. Does not perform any of the error checks of PACKDT, so will produce garbage if given garbage.

CONVUDT(strg, uv) - performs the same conversion as WRITEUDT, but deposits the result in the string strg. Returns strg as its value.

FINDMONTH(strg) - tries to interpret the string strg as the name of a month. If successful, returns the month number (0 through 11); if unsuccessful, returns -1. Strg must be at least 3 characters long, and must be the prefix of some month name, ignoring upper/lower case distinctions.

MONTHNAME(mo) - returns a string which is the name of month mo (0 through 11), e.g. "December". The user should not write into this string.

WEEKDAY(dv) - returns the day of the week of dv (Monday=0, Sunday=6).

2. Timer

The timer package is written in assembly language. It is found in TIMER.BR. It defines 3 procedures (TIMER, SETDAYTIME, DAYTIME) and does not require any external procedures. It does use 6 locations in page 1, currently 572 through 577, which are permanently reserved for it. The procedures perform the following functions.

TIMER(tv) - reads the millisecond timer into tv!0 and tv!1. Returns tv!1 as its value. This function is available as part of the Alto operating system.

SETDAYTIME(dv) - declares the current time-of-day to be the time-of-day in dv!0 and dv!1. (This value might have been constructed using the PACKDT procedure in the daytime package. It is not reasonable to compute time-of-day values by hand.) This function is available as part of the Alto operating system.

DAYTIME(dv) - reads the current time-of-day into dv!0 and dv!1. Returns dv as its value. This function is available as part of the Alto operating system.

2.1. UPDATETIMER

The timer package uses an auxiliary procedure UPDATETIMER(), found in UPDATETIMER.BR, to move timing information from the hardware clock into software variables. Since this procedure must be called at least once a second (on the average) for the timer package to function properly, the operating system calls UPDATETIMER() on every display field interrupt. The timer package also calls UPDATETIMER under some exceptional circumstances (turning the interrupt system off during the call), so UPDATETIMER must be loaded to use TIMER. User programs should not call UPDATETIMER at all.

O.S. maintainers note: the page 1 pointers in UPDATETIMER.A must agree with those in TIMER.A, otherwise there will be chaos.

Procedure tracing package

A package is now available for tracing BCPL procedures on the Alto, similar to the TRACE facility available in Interlisp. The tracing package is available as TRACE.C, a BCPL source program. To use it, you will normally also need the formatted output package, available as FORMAT.SR, also BCPL source.

The tracing package supplies six procedures (TRACE, PTRACE, UNTRACE, TRACETABIN, TRACEWS, TRACEPUTS) and two statics (TRACESTREAM, TRACELINES). TRACE(proc, str) turns on tracing of procedure proc; UNTRACE(proc) turns off tracing of proc. PTRACE(proc, tproc) turns on tracing of proc, but instead of using str to construct a message as described below, it calls tproc before entering the body of proc, as

```
        tproc(proc, lv arg0, n, 0)
```

where n is the number of arguments and arg0 is the first argument; when proc returns, the tracer calls

```
        tproc(proc, lv arg0, n, lv val)
```

where val is the value returned. (Note that tproc may alter the arguments or the return value if it wishes.) Proc may be any BCPL procedure (including the procedures in the TRACE and FORMAT packages), or any assembly language procedure that begins with the same 4 instructions as a standard BCPL procedure, i.e.

```
        STA 3,1,2
        JSR @370
        frame size
        JSR @367
```

All output produced by tracing goes to the stream TRACESTREAM; if TRACESTREAM=0, output goes to the system display stream DSP. If TRACELINES is non-zero, the tracer will pause after every TRACELINES lines of its own output, as follows:

```
        print 3 *'s;
        waiting for a character to be typed;
        print 2 more *'s;
```

before proceeding. Note that other output to the same stream (e.g. from the program being traced) is normally not counted, since the tracer can't intercept it. However, programs such as the procedures supplied as the tproc argument for PTRACE may take advantage of the pause feature by using TRACEWS(string) or TRACEPUTS(char) to do their output: aside from line counting, these are equivalent to WSS(TRACESTREAM, string) and PUTS(TRACESTREAM, char).

The output produced for a TRACEd procedure consists essentially of the arguments when the procedure is entered, and the value when the procedure returns. Output is indented 2N mod 16 spaces, where N is the depth of nesting in traced procedures, similar to the Interlisp convention. (The procedure TRACETABIN() writes the appropriate number of spaces on TRACESTREAM.) The format of the output is determined by the str argument to TRACE. There are 4 cases:

1) Str=0, or str omitted, e.g. TRACE(foo). In this case, the message on entry is

```
        locfoo:
        arg1 arg2 ... argn
```

where locfoo is the octal location of the first instruction of foo, and the arguments are printed in octal (by WOS). The return message is

```
                locfoo returns val
```

where val is the value returned, also in octal.

2) Str contains neither $; nor $:, e.g. TRACE(foo, "Foo"). The messages are the same, except that the string Foo appears in place of the location locfoo.

3) Str contains a $;, e.g. TRACE(foo, "foo: a1=<D>;foo = <B>"). In this case, the portion of str before the $; is used as the format for printing the arguments, and the portion after the $; is used for printing the value. If there are more arguments than <> fields, the extra arguments are printed with WOS; if there are fewer, printing stops after the last <> field for which an argument was supplied. This produces pleasing output for procedures which take variable numbers of arguments.

4) Str contains no $;, but does contain a $:, e.g. TRACE(foo, "FOO: A1=<D>"). This is equivalent to TRACE(foo, "FOO: A1=<D>;FOO returns <B 6>"), i.e. the string up to the $: is taken as the procedure name and the word "returns" and an octal format are supplied.

Of the 4 options, 1 and 2 do not require the presence of the FORMAT package; 3 and 4 do require FORMAT if str contains any <> fields. In the latter case, if the FORMAT package is not loaded, all values will be printed with WOS. Use of PTRACE does not require the FORMAT package, unless, of course, the user's own trace-print procedures use FORMAT.

Note that TRACE can be called from SWAT, but only with str omitted or zero. PTRACE and UNTRACE can be called freely from SWAT.

Alto virtual memory

A package is now available which provides a virtual memory facility for Alto programs. The virtual address space is 2↑↑24 16-bit words; the page size is any power of 2 times the disk sector size (256 words).

The VMEM package uses several data structures for which you (the user) must supply storage, as follows:

1) A hash map, whose size is 2P+3 words, where P is the largest number of 256-word paging buffers you will ever have allocated at one time, rounded up to a power of 2 (e.g. if you have 20K for paging buffers, this is 80 buffers, so P=128).

2) A buffer pointer table of 256 words.

3) Paging buffers, as many as you want, located anywhere in core (not necessarily contiguous). Each group of buffers is truncated if necessary so that it starts on a multiple of the page size and is a multiple of the page size long.

4) A locked cell list, whose size is the largest number of cells you will ever want to use as locks (see below).

5) Two statistics tables, of 100b words each (optional).

The code comes in a fair number of pieces. It uses BFS.C, DVEC.C, and BFSML.A from Butler's basic file system, which is in <ALTO>BFS.DM. The other pieces are in <ALTO>VMEM.DM. VMEM.C is the main logic. ASMAP.A is a small assembly code package that actually invokes the mapping microcode. If you don't have a RAM, you need SOFTMAP.A in addition, which patches into the emulator trap vector.

****************** CAVEATS ******************

In the version of VMEM currently released, page groups are not implemented (though variant page sizes are implemented). This will be rectified eventually.

The procedure for getting the relevant microcode into a RAM and getting it properly hooked up to the Nova emulator has not been worked out. The best current solution (being used for Lisp) requires changes to the standard Alto and is not recommended for the world at large.

1. Initialization

INITIALIZEVMEM(HMAP, HMAPSIZE, BPTAB, LCL, LLCL, PGSIZE)

HMAP is the address of the hash map; HMAPSIZE is 2P (256 in the example of 80 buffers.) (VMEM will clear the hash map.) BPTAB is the address of the buffer pointer table. LCL is the address of the locked cell list, and LLCL is its length. PGSIZE is the page size (400b, 1000b, etc.).

Before doing any mapping operations, you may initialize PAGESTATS and MAPSTATS to the origins of the two statistics vectors; if you don't, you just won't get any statistics. You must clear these vectors yourself.

INITSOFTMAP()

If you don't have mapping microcode, you must call INITSOFTMAP after calling INITIALIZEVMEM.

ADDBUFFERS(FIRST, LAST)

In order for the mapping routines to function, you must give them space for page buffers with ADDBUFFERS. FIRST and LAST are the bounds of a core area to be used for this purpose. FIRST will be rounded up to the next multiple of the page size if necessary, and LAST+1 rounded down; thus ADDBUFFERS(7700b, 10077b) followed by ADDBUFFERS(10100b, 10377b) will NOT result in the space from 10000b through 10377b being made into a page buffer.

2. Mapping functions

A 24-bit address:
```
        $-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$
        |   high part   |              low part         |
        $-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$
        |        virtual page part      |   word part   |
        $-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$-+-+-+-+-+-+-+-$
```

"The virtual address (HI, LO)" means a virtual address whose high part is bits 8-15 of HI (bits 0-7 being zero) and whose low part is LO. Note that the "page size" referred to above ONLY affects the unit of transfer to and from the disk: "virtual page part" ALWAYS refers to the high 16 bits of a 24-bit address.

All of the mapping functions described in this section are declared global (page zero), so you must declare them external with @-sign.

VRR2(HI, LO)

Returns a core address corresponding to the virtual address (HI, LO), having read the page into a buffer if necessary.

VWR2(HI, LO)

Same as VRR2, but assumes you are about to write into the page, so marks it as needing to be rewritten onto the disk.

VRR1(LO)

Same as VRR2(0, LO). If you only have a 2↑↑16-word virtual space, you can save a small amount of code by using VRR1 instead of VRR2.

VWR1(LO)

Same as VWR2(0, LO).

VRR(PTR)

Same as VRR2(PTR!0, PTR!1). Useful if you are carrying around addresses in vectors, as Lisp does.

VWR(PTR)

Same as VWR2(PTR!0, PTR!1).

VRRP(VP)

Same as VRR2(VP RSHIFT 8, VP LSHIFT 8), i.e. converts a virtual address whose virtual page number is VP and whose word part is zero. Useful if you are only using the virtual memory package to manage buffers, and doing your own data scanning.

VWRP(VP)

Same as VWR2(VP RSHIFT 8, VP LSHIFT 8).

3. Statistics

PAGESTATS is a table of 20b pairs of 32-bit counters. Letting G be the high 4 bits of the 24-bit address, entry 2G (words 4G and 4G+1) in PAGESTATS is incremented for each disk read, and entry 2G+1 (words 4G+2 and 4G+3) is incremented for each write. MAPSTATS is a similar table, incremented for each mapping reference.

If you use SOFTMAP, NPROBE points to a 32-bit counter of the number of secondary probes in the hash map.

4. Other facilities

REHASHMAP(VP)

Looks up the virtual address VP*400b in the hash map, returning 0 if present, or the address of an appropriate empty slot if not present. Used by the page fault routine to reconstruct the hash map, but also useful for determining quickly whether a page is in core.

SNARFBUFFER(BUFPTR)

BUFPTR must be the address of a buffer (i.e. a multiple of the page size) within the scope of some previous call to ADDBUFFERS. The effect of SNARFBUFFER is to remove that buffer from use by the virtual memory package, for example if you want to use it to hold display data.

If the buffer is locked (see below), SNARFBUFFER returns 0; normally SNARFBUFFER returns the address of the buffer. If you don't care what buffer you snarf, BUFPTR=0 will select an arbitrary non-locked buffer.

UNSNARFBUFFER(BUFPTR)

Reverses the action of SNARFBUFFER.

LOCKCELL(LVLOCK, RELOC)

Declares that the cell whose address is LVLOCK holds a core address which must remain valid across page faults, i.e. the buffer in which it lies must not be re-used. Note that the extra level of indirection means that your program can store into the lock cell freely. As a consequence, if you store some arbitrary bit pattern into a lock cell, it will function as a lock if it happens to constitute an address within some buffer.

If RELOC is false or absent, the buffer pointed to from rv LVLOCK will not be swapped or moved, nor can it be snarfed. If RELOC is true, the contents of the buffer may be moved to another buffer (if the buffer is snarfed, or if this is necessary to be able to read in a page group), and the pointer will be fixed up.

The number of different lock cells is limited to the parameter LLCL supplied to INITIALIZEVMEM. If the lock list is full, LOCKCELL calls SWAT.

UNLOCKCELL(LVLOCK)

Undoes the action of LOCKCELL. Returns true if LVLOCK was actually in the lock cell list, or false if it was not.

ISLOCKED(PTR, RELOC)

If PTR is a pointer into a locked buffer, returns the address of the entry in the lock cell list which points to the relevant lock cell; if not, returns 0. If RELOC is absent, considers all lock cells; if RELOC is true, considers only relocatable locks; if RELOC is false, considers only non-relocatable locks.

FLUSHBUFFERS()

Rewrites all dirty pages from buffers onto the disk, including locked pages, and generally tidies things up in preparation for quitting. (It is OK to go on using the virtual memory after this, you just have to do another FLUSHBUFFERS before quitting eventually.)

DOUBLEADD1(PTR)

Adds 1 to (PTR!0, PTR!1) as a 32-bit number.

5. User routines

The VMEM package makes no assumptions about the correspondence between virtual addresses and disk pages. In fact, VMEM does not even assume that the swapping is done on the disk -- you can use the Ethernet or magtape if this suits your fancy. Consequently, you must supply a number of routines to establish this correspondence.

Remember that VPAGE, below, always refers to 256-word pages.

CHECKVPAGE(VPAGE, WFLAG)

This routine is called on a page fault to determine if a page has never been referenced, already exists, or is invalid. VPAGE is a virtual page number (the high 16 bits of a 24-bit address); WFLAG is true if the fault was from a write reference, false if from a read reference. CHECKVPAGE must return true if the page is a new page, false if the page already exists. If the page is invalid, CHECKVPAGE can do whatever it wants, but it should not return.

DOPAGEIO(VPAGE, CORE, NPGS, WFLAG)

This routine must transfer NPGS 256-word pages, starting at virtual page VPAGE and core address CORE, to or from the swapping medium, depending on WFLAG: false means read, true means write.

PAGEGROUPBASE(VPAGE)

PAGEGROUPSIZE(VPAGE)