ALTO SOFTWARE PACKAGES

Compiled on: October 16, 1977

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

This list is a directory of major Alto software packages. The files for these programs are available on the <ALTO> directory. The documentation for these packages is available on <ALTODOCS>. This document is filed as <ALTODOCS>PACKAGES.EARS. Some packages have closely-corresonding subsystems (e.g., TFS Trident disk software and TFU utility); in this case, the bulk of the documentation is located with in the Alto Subsystems Manual, and a cross-reference is included in this document.

The items listed below may be flagged by a single character to indicate where the documentation may be found:

* documentation for these items is contained within this manual;
** these items are described further in a separate document;
# see the Alto Operating System manual for documentation.

#ALLOC: A boundary-tag storage allocator. Documentation is in the Alto Operating System Manual. (Ed McCreight)

*ASIM: A procedure which simulates an Alto microprocessor equipped with a RAM. (Peter Deutsch)

*BCPLRUNTIME: A replacement for the standard Bcpl runtime (in the OS), in whihcch nearly all of the operations have been microprogrammed. Typical Bcpl programs run 25 to 30 percent faster. (Ed Taft)

#BFS: The "basic file system" subroutines. These do page-oriented I/O to disk files organized according to standard Alto conventions. Documentation is in the Alto Operating System Manual. (Butler Lampson)

*BYTEBLT: transfers an arbitrary block of 8-bit bytes from one place in memory to another. (Ed Taft)

*CMDSCAN: an interactive command scanner and collection of command interpretation procedures. (Ed Taft)

*CONTEXT: provides facilities for managing multiple execution contexts for Bcpl procedures. (Ed Taft)

DCBPRESS: This file provides one subroutine for making a one-page Press file from an Alto screen bit-map. The calling sequence is: DCBPress("filename", pDCB, [width, height, left, top]), which writes a file of the given name using pDCB as a pointer to a display control block.. The last four parameters allow you to select a portion of the rectangle described by the DCB for printing. Width is the width (in bits) of the window you wish to see; height is the height in scan-lines; left is the offset from the left edge of the bit-map; top is the offset from the top of the bit-map. (Bob Sproull)

*DIABLOPRINTER: Routines that implement streams on the Diablo printer. (Ed Taft)

**DIALOG and DIRECTOR: a set of run-time subroutines for conducting a dialog between a human and a program, and a mechanism for controlling the execution of a 'batch' of programs. Further documentation is on <altodocs>Dialog.ears. (Jay Israel)

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

#DSTREAM: Provides a capability for display streams, multiple fonts, bit repositioning, selective erasing and polarity inversion. Documentation is in the Alto Operating System Manual. (Peter Deutsch)

*EFTP package: A Pup-based file-transfer package using a simple (EFTP) communications protocol. (John Shoch)

*ETHERBOOT: A subroutine that will "boot" the Alto from one of several boot files supplied by Ethernet gateways. (Ed Taft)

*EVENTREPORT: A subroutine that will log events using the Pup Event Report protocol. (Bob Sproull)

*FINDPKG: searches standard Alto files for certain simple kinds of patterns at very high speeds using special microcode. (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)

*FTPPACKAGE: File Transfer Protocol (FTP) routines. (David Boggs)

*GETSETBITS: makes it easy to extract and replace strings of up to 16 bits in a vector of bits. (Peter Deutsch)

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

*INTERRUPT: permits Bcpl procedures to be called as a result of hardware interrupts on the Alto. (Ed Taft)

*ISF: a package that provides pseudo-ramdom access to Alto files. (Peter Deutsch)

*KBD: provides a basic keyboard input stream capability. (Peter Deutsch)

*KPM: a simple efficient Knuth-Pratt-Morris pattern match of a name against a template that may contain one or more wildcard characters. (Ed Taft)

*LOADRAM: loads a 'Pack Ram Image' (see PackMu in the subsystems documentation) into the ram, and optionally performs a 'silent boot' to start one or more tasks in the Ram. (Ed Taft)

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

*OVERLAYS: Subroutine package for handling Bcpl overlays conveniently. (Peter Deutsch)

*PAPERTAPE: A package which implements streams to a high paper tape reader and punch which can be attached to the Alto via the Diablo printer interface. (David Boggs)

*PUPERPSERV: implements the Pup Event Report Protocol (ERP). (David Boggs)

*PUP PACKAGE: implements communications by means of Pups and Pup-based protocols. (David Boggs and Ed Taft)

*QUEUE: a simple set of queue primitives. (Ed Taft)

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

*READUSERCMITEM: reads items from user profile files. (Peter Deutsch)

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

*RENAMEFILE: renames a file. (David Boggs)

*RINGBUFFER: a set of procedures for buffering data by means of circular buffers. (Ed Taft)

*RWREG: Procedures for reading and writing the Alto microprocessor R and S registers under program control. (Peter Deutsch)

*SCANFILE: This package provides procedures for reading Alto files at full disk speed, and overlapping computation with the reading. (Peter Deutsch)

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

*SDIALOG: A package for managing simple interactive dialogs with a user. It helps prompting and response parsing. (Bruce Parsley)

**SORTPKG: a package for sorting things of arbitrary sort--you provide a "get" routine, a "put" routine and a "comparison" routine. Documentation is found in the first page of the Bcpl sources. (Ed McCreight)

*SPLINE: procedures for fitting cubic splines to sets of knots. (Patrick Baudelaire)

#STREAMS: The disk streams package provides facilities for doing efficient sequential input/output to and from Alto disk files. Documentation is in the Alto Operating System Manual.

*STRINGS: useful procedures for extracting, concatenating, and comparing strings, plus string streams (Ed Taft)

*TEMPLATE: formats output to a stream according to a template provided as a string. (Ed Taft)

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

*TIMER: a set of procedures for setting, testing and blocking on timers. (Ed Taft)

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

*UTILSTR: A collection of utility and string-manipulation procedures. (Bruce Parsley)

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

Alto processor simulator

The Asim library package very precisely simulates the Alto I or II processor, including the 2K ROM and extended memory options. All references to the various Alto memories (R registers, microinstruction ROM/RAM, constants, main memory) occur through procedures, so the simulator may be run using the actual contents of the RAM or a core image thereof, the real contents of main memory or a SWAT-like file image thereof, and so on. Memory timing is simulated properly, and a large number of minor logical errors (such as mis-timing of memory references, assuming that L or T is safe over a TASK, or giving a branch modifier in the instruction after a TASK) are detected.

1. Requirements

Asim expects the user to provide the following 7 procedures (and declare them external):

ReadR(j) - return the contents of the j'th R register. J may be 0 through 37B or 41B through 77B.

WriteR(j, wd) - write the value wd into the j'th R register.

ReadRAM(j) - read a word from the instruction memory as described in the Alto reference manual, to wit: bit 4 of j decides between ROM (1) and RAM (0); bit 5 of j decides between upper 16 bits (1) and lower (0); bits 6-15 of j give the address. If Asim is simulating an Alto (II) with the 2K ROM option, then when bit 4 of j is set, bit 3 of j chooses between ROM0 (0) or ROM1 (1). Note that this is not supported by the actual Alto hardware.

WriteRAM(j, wd) - write wd into the instruction memory. J is as for ReadRAM. Note that unlike the hardware instruction, this procedure must be capable of writing into the upper and lower 16 bits independently.

ReadCON(j) - return the j'th constant. J is between 0 and 377B.

ReadMEM(j) - return the contents of main memory location j. If Asim is simulating an Alto (II) with extended memory, it will normally call ReadMEM(j, bank), where bank provides the 2 extra bits of memory address. ReadMEM will still be called sometimes with only one argument for accessing I/O locations (177000B and above), and it should check for j in this range before examining bank. Note also that, as for the real extended memory hardware, Asim uses the contents of (simulated) location 177740B to determine the bank numbers for all memory accesses.

WriteMEM(j, wd) - write wd into main memory location j. With extended memory, Asim calls WriteMEM(j, wd, bank) -- note the order of the arguments.

The user program may use any implementation it wishes for these operations. The only requirement is consistency, i.e. a Read operation must retrieve the datum given to the last Write operation for that cell.

Either the READMU package, or the PACKMU subsystem and ReadPackedRAM package, described in separate writeups, may be useful for reading microcode into memory for simulation.

2. Use

Asim is written in Bcpl and consists of a single file Asim.BR. It does not use any facilities of the Alto OS. It provides two externally accessible procedures (InitAsim, Asim) and a large number of externally accessible statics. The procedures and accessible statics are declared external in the file Asim.D which the user should "get".

InitAsim(altotype, extrarom, extendedmemory) initializes the simulator state completely -- declares the main memory interface to be quiescent, clears all internal registers to zero, and marks L and T as undefined. It does not affect any of the memories. Altotype (defaults to 0) specifies the Alto configuration: 1 means Alto I, 2 means Alto II, and 0 means that the microprogram is supposed to execute compatibly on both Alto I and Alto II. Extrarom (defaults to false), if true, means that the Alto has the 2K ROM option. Extendedmemory (defaults to false), if true, means that the Alto has the extended memory option.

Asim() executes one micro-instruction. Asim returns 0 if the instruction completed successfully, otherwise a string which indicates the reason for the failure. In the latter case, no change has occurred in any programmer-visible state (R, RAM, main memory, L, T, IR, carry flags, etc.), offering the possibility of repairing a problem and resuming execution.

Asim maintains the state of the microprocessor in a set of Bcpl statics which are available to the user for inspection. These statics are supposed to capture the entire program-visible state of the microprocessor, plus a few useful quantities which are not normally visible from the outside between instructions. The caller of Asim is free (but not encouraged) to alter any of Asim's accessible statics between instructions -- there are no hidden interactions. The accessible statics are documented in Asim.D.

2.1. Errors detected

The following is a (currently) complete list of the strings which Asim will return.

```
L undefined
T undefined
Branch modifier following TASK
Delayed F1 following TASK
TASK with memory running
ALU output discarded
DNS with BS#R←
2 memory ops
Memory timing error
Attempt to load R40
Attempt to mask MD
Bad ALUF
MAR← with R37
STARTIO
Bad F1
Bad F2
Attempt to shift into 2nd R bank
MD← at wrong time
←MD at wrong time
Odd double fetch not compatible
```

2.2. Limitations

Asim only simulates single and double fetches and single stores to main memory, not the more exotic types of memory access. This is not an intrinsic limitation and could be fixed if there were enough demand for it.

Asim only simulates the emulator task.

The following is a listing of the current contents of Asim.D.

```
//
// External definitions for Asim
// last edited  October 6, 1977   10:25 AM
//

external        // entry points
[       InitAsim        // (altotype [1], extrarom [false], extramemory [false])
        Asim    // () -> 0/errorstring
]

external        // the microprocessor state
[       @t      // T
        @tu     // T undefined flag (true or false)
        @l      // L
        @lu     // L undefined flag (true or false)
        @ir     // IR
        @carry  // emulator carry (0 or 1)
        @bus    // temp. for bus data
        @alu    // temp. for ALU output
        @sh     // temp. for shifter output
        @skip   // SKIP (0 or 1)
        @alucy  // last ALU carry (0 or 1)
        @mar    // last memory address
        @altbank        // true iff last MAR← selected alternate bank
        @mstate // memory state
        @nmod   // modifiers for NEXT
        @pc     // (microinstruction) PC
        @waiting// TASK, RDRAM, WRTRAM, SWMODE waiting or -1
        @ramadr // RAM address
]
```

Bcpl Runtime Package

This package is a replacement for the standard Bcpl runtime (the one built into the Alto Operating System), in which nearly all of the operations have been microprogrammed. Typical Bcpl programs run 25 to 30 percent faster than with the standard routines, depending primarily on their frequency of procedure calls and their richness in complex structure references. Use of this package also permits one to Junta to levBasic if desired, for a savings of approximately 500 words of main memory.

The microprogrammed runtime is entirely compatible with the standard one. It does not require programs to be modified or recompiled, and it works correctly during calls to the Operating System as well as to your own procedures. The simplest use of this package requires only that you load the necessary microcode into the Ram and call one initialization routine.

The package also provides a convenient framework in which to define and microprogram additional emulator opcodes.

1. Standard Use

The simplest case applies when you do not need to include any special microcode of your own. The file BcplRuntime.Dm is a dump-format file containing BcplRuntime.Br and BcplRuntimeMc.Br. These modules should be loaded with your program, along with the LoadRam procedure, available separately as LoadRam.Br.

Early during initialization, your program should execute the following:

```
external [ LoadRam; InitBcplRuntime; RamImage ]
if LoadRam(RamImage) eq 0 then InitBcplRuntime()
```

(LoadRam returns zero if it successfully loaded the Ram and a nonzero result otherwise, e.g., because no Ram board is installed.)

Once this has been done, the space occupied by LoadRam.Br and BcplRuntimeMc.Br may be reclaimed. BcplRuntime.Br must remain resident throughout execution of the program, but it occupies only about 150 words whereas the others consume nearly 3000.

InitBcplRuntime sets up a 'user finish procedure' (in the manner described in the O.S. manual, section 3.12), whose purpose is to restore the normal Bcpl runtime routines when the program 'finish'es for any reason. Operation of this mechanism is ordinarily invisible; however, there are two situations in which the programmer must be aware of its workings.

First, if you execute a Junta and later a CounterJunta, the CounterJunta will itself cause the standard Bcpl runtime to be restored. The later restoration performed by the BcplRuntime package will be redundant and will do no harm, but the standard (slower) Bcpl runtime will be in use once the CounterJunta has been executed.

Second, if you Junta away the standard Bcpl runtime routines themselves, you must be careful to perform initialization in the correct order. In particular, InitBcplRuntime must be called before the Junta and before any other code that sets up user finish procedures. This ensures that at 'finish' time, the cleanup procedure in the BcplRuntime package will be the last user finish procedure executed, immediately before control returns to the operating system for the final time. If this convention is not followed, a subsequent call on the Bcpl runtime would end up diving into garbage (since InitBcplRuntime saves and restores only the runtime statics, not the code).

2. Adding Your Own Microcode

In order to implement additional emulator instructions or install microcode for special devices, it is necessary to understand the workings of the package in some detail. If you don't want to do those things, you need read no further.

The source files are contained in the dump-format file BcplRuntimeSource.Dm. It includes, among other things, the following microcode source files:

BcplRuntimeMc.Mu	The top-level microcode source file, which 'includes' all the others.
EmulatorDefs.Mu	Standard label and R-register definitions useful in writing code to be run as part of the emulator task.
RamTrap.Mu	Declarations and code for dispatching all opcodes that trap into the Ram.
GetFrame.Mu	Microcode implementing the Bcpl runtime 'GetFrame' and 'Return' operations.
BcplUtil.Mu	Microcode implementing all remaining Bcpl runtime operations.

In addition to these files, you need AltoConsts23.Mu (or whatever the current version is), Mu.Run, and PackMu.Run. The latest (October 11, 1977) version of Mu is required.

To add new opcodes, you will need to edit BcplRuntimeMc.Mu and RamTrap.Mu (which should be renamed to something else first). The changes to BcplRuntimeMc.Mu are trivial: simply append 'include' statements for each of your own source files.

RamTrap.Mu contains the following predefinition:

```
    !37,40, TrapDispatch,,, GetFrame, Return, BcplUtility;
```

The labels in this predefinition correspond to the opcodes #60000, #60400, #61000, #61400, ..., #77400 (a total of 32). However, several of these cannot be used because their execution does not cause a trap into the Ram. These are #60000, #60400, #61000, #64400, #65000, #67000, and #77400. The GetFrame, Return, and BcplUtility instructions use #61400, #62000, and #62400. All others are available for your own use simply by adding labels to the predefinition.

When one of these labels is reached, the Alto is in a clean state (no TASK or memory reference pending), the accumulators AC0 through AC3 contain the values supplied by the emulated program, and IR (the DISP bus source) contains the low-order 8 bits of the opcode, which may be used for further dispatch if desired.

The routine should finish by executing the following sequence of operations:

```
TASK;
something;
SWMODE;
:START;
```

It is essential that the TASK be executed as late as possible before the branch to START. The worst-case path in the Rom microcode beginning at START consists of 19 microinstruction cycles without a TASK. It has been determined empirically that as few as 3 microinstructions inserted between 'something' and 'TASK' in the above sequence causes Diablo Model 44 disks to get data-late errors. (Alas, it is not possible to say 'SWMODE, TASK' in one microinstruction because they are both F1's. In hindsight, it would have been nice if SWMODE had been implemented in such a way as to cause a TASK also.)

BcplUtil.Mu contains three convenient exit points to which opcode emulation routines may branch. The code for these exit points is:

```
Start0: PC←L;
Start1: L←PC, SWMODE;
Start2: PC←L, :START;
```

One may branch to Start0 having just executed 'L← new PC, TASK;', to Start1 having just executed 'TASK; something;', or to Start2 having just executed 'TASK; something; L← new PC, SWMODE;'.

Standard R-registers available to the routine are listed in EmulatorDefs.Mu. These are SAD, XREG, XH, MTEMP, DWAX, and MASK. All except MTEMP are used exclusively by the emulator task and may be clobbered arbitrarily (the standard Nova emulator in the Rom does not depend on them). MTEMP is usable by any task but is safe only until the next TASK.

You may need to modify EmulatorDefs.Mu if your microcode defines labels in low, fixed locations (e.g., START or the task starting addresses). Note that EmulatorDefs.Mu defines all labels except TRAP1 in a way that does not consume space in the Ram. You may need to change one or more of these (e.g., START) to ordinary predefinitions if you intend to define them in the Ram.

The microcode is assembled and turned into a .Br file by means of the commands:

```
Mu BcplRuntimeMc.Mu
PackMu BcplRuntimeMc.Mb BcplRuntimeMc.Br
```

The Bcpl runtime microcode contained in the package occupies 337 (decimal) microinstruction words.

ByteBlt -- Fast Byte Block Transfer

This package contains a single procedure, ByteBlt, which transfers an arbitrary block of 8-bit bytes from one place in memory to another as quickly as is possible without special microcode. The procedure handles all cases of blocks starting or ending on even or odd byte boundaries and whose lengths are even or odd. The bulk of each transfer is done using the "blt" instruction if possible and using a fast inner loop (4 instructions per byte) otherwise.

ByteBlt is written in assembly language. It is distributed as AltoByteBlt.br, which is assembled from AltoByteBlt.asm. It is 107 (decimal) instructions long and calls no external procedures (aside from the BCPL runtime). A Nova-compatible version of this package is also available (though it works less efficiently due to lack of a "blt" instruction).

ByteBlt(DstAdr, DstByte, SrcAdr, SrcByte, ByteCount)

> Transfers the block of bytes described by the arguments. Bytes are packed two per word, with the left byte considered to be the first. DstAdr and DstByte specify the destination address of the first byte, with DstAdr providing a base word address and DstByte specifying the offset of the first byte relative to that word (0 means the left byte in DstAdr, 1 means the right byte in DstAdr, 2 means the left byte in DstAdr+1, etc). Similarly, SrcAdr and SrcByte specify the source address of the first byte. ByteCount is the number of bytes to be transferred (must be less than 2↑15; zero is legal).
>
> No bytes outside of the specified destination block are affected; in particular, if the destination block begins on a right-hand byte or ends on a left-hand byte, the other byte in the same word is not clobbered. However, the source and destination blocks must not overlap.

ByteBlt achieves its efficiency by checking for three special cases. If the block is very small (4 bytes or less), it is transferred by means of a relatively slow byte-at-a-time routine, since the overhead of setting up for the other, faster cases outweighs this inefficiency. (However, this case is still much faster than moving bytes using a BCPL "for" loop and structure references).

If the source and destination blocks are in phase (i.e., they both start with the left byte or both with the right), then the entire block, possibly excepting the first and last bytes, is transferred by means of a single "blt" instruction. Leftover bytes at either end are handled specially.

If the source and destination blocks are out of phase, then the bytes are transferred by means of a 16-instruction inner loop which reads and writes data in memory two full words at a time, swapping and masking bytes as required.

Command Scanner Package

This package consists of an interactive command scanner and a collection of command interpretation procedures. Among the important features of this package are:

1. The editing facilities are fairly sophisticated. One can provide defaults and modify the break and echo sets on a per-phrase basis. The user is permitted to backspace over phrases that have already been parsed. Phrases may be interspersed with "noise" text that is retained with the command line while not logically a part of it.

2. Error recovery and retry facilities are provided by means of some rather tricky BCPL control structure.

3. The package is modular, and not all modules necessarily need be loaded. Also, specialized knowledge about the Alto display is confined to one module, which may be replaced by a different module that deals with other media such as hardcopy terminals or network streams.

The Command Scanner Package is intended for use in programs with relatively sophisticated needs, and is fairly large (just the basic command editor and Alto display handling modules together amount to about 1500 words of code). Programmers with simpler needs or tight memory constraints might be better off using Bruce Parsley's Simple Dialoging Package.

1. Organization

The package is distributed as a dump-format file CmdScan.dm, which contains the following files:

CmdScan.decl	Declarations that may be needed in order to use the package.
CmdScan.br	The main control module. This must always be loaded.
CmdScanEdit.br	Editing operations invoked from the main control module. This also must always be loaded.
CmdScanDisplay.br	Operations specific to the Alto environment (display and keyboard). This or some equivalent module must always be loaded.
CmdScanTty.br	Equivalent operations oriented toward a minimal terminal stream interface.
CmdScanAux.br	Higher-level command interpretation procedures for dealing with such things as numbers, strings, filenames, and keywords. This module is required only if its facilities are desired.
Keyword.br	Primitives to look up and enumerate keywords in a keyword table. Procedures in this module are called from the CmdScanAux module.

KeywordInit.br	Procedures to construct and manipulate keyword tables. This module may be discarded after all desired keyword tables have been created.
CmdScanOEP.br	Declarations of Overlay Entry Points (OEPs) in the CmdScan modules. This module is needed only if the CmdScan modules are loaded into overlays.
KeywordOEP.br	OEP declarations for the Keyword modules.

The CmdScanAux module requires that the Timer and Context packages also be loaded. If one is not using contexts, one may omit the Context package and instead define an external procedure Block() that just returns immediately.

2. Basic Command Scanner

Command scanning is done within the confines of a Command State (cs) object, which accumulates the text of a command and maintains state from one phrase to the next. A command consists of a sequence of phrases, possibly interspersed with "noise" text not part of any phrase. Each phrase consists of zero or more non-terminating characters followed by a terminating character.

Editing is done on a phrase-by-phrase basis. For each phrase, the GetPhrase procedure is called to input a new phrase from the keyboard (terminated by a break character) and append it to the command line. GetPhrase returns when the terminating character is typed. At this point, the caller may call Gets(cs) to read the characters of the phrase (using Endofs(cs) to test for end of phrase).

While control is inside GetPhrase, if the user backspaces past the beginning of the current phrase, control is sent back to an earlier point of interpretation so as to reparse the previous phrase now being editted. There exists a facility for regaining control when this happens so as to release resources acquired during command interpretation.

Between phrases, one may output "noise" text by means of Puts(cs). This text is displayed and maintained in the command line but does not participate in editing operations. That is, if one is positioned at the end of a "noise" string and backspaces one character, the entire "noise" string is erased along with the real command character preceding it.

2.1. Getting Phrases

The following procedures are defined in CmdScan.br and CmdScanEdit.br:

InitCmd(maxChars, maxPhrases, WordBreak [DefBreak], PhraseTerminator [DefBreak], Echo [DefEcho], keyS [keys], dspS [dsp], Erase [DefErase], Error [DefError], zone [sysZone]) = cs or 0

Creates and returns a Command State (cs) structure capable of holding at most maxChars characters grouped into at most maxPhrases phrases. keyS and dspS are the keyboard and display streams for the command scanner. The structure is allocated from zone. The remaining arguments (all of which are procedures) control the command scanner in various ways. These procedures are described below under "Edit Control Procedures".

When InitCmd is called, it always returns a cs. However, if the command is deleted (by the user striking the Delete character) during later command typein, the cs is destroyed and InitCmd returns again with zero as its result. This is discussed below under "Backing Up and Catch Phrases".

Closes(cs)
Destroys the Command State structure cs, returning it to the zone from which it was allocated.

GetPhrase(cs, WordBreak [default], PhraseTerminator [default], Echo [default], Help [], helpArg []) = numChars
Readies the next phrase to be interpreted, inputting one from the keyboard if necessary. Returns the number of characters in the phrase, not including the terminating character.

The WordBreak, PhraseTerminator, and Echo procedures, if provided, override the ones declared to InitCmd for this phrase only. If Help is provided then upon typein of a question mark the call Help(dspS, helpArg) is executed; this is expected to output a helpful message to the stream dspS, not preceded or followed by a carriage return. (Typically the message is just a string, which may most easily be output by providing a Help procedure of Wss and a helpArg of the string itself.)

Gets(cs) = char
Returns the next character of the current phrase, i.e., the one most recently input by means of GetPhrase. If the phrase is exhausted (i.e., the next character would be the phrase terminator), Errors(cs, ecEndOfPhrase) is called.

Endofs(cs) = true|false
Returns true if the current phrase is exhausted.

Puts(cs, char)
Appends the "noise" character to the command line and outputs it to the command's display stream. Puts should be called only between phrases, i.e., after reading all characters of one phrase and before calling GetPhrase for the next.

Resets(cs)
Resets the command scanner to the beginning of the current phrase, such that the next call to GetPhrase will return the same phrase again.

TerminatingChar(cs) = char
Returns the character that terminated the current phrase.

2.2. Default Phrases

DefaultPhrase(cs, string, char [])
Supplies a default value (the string) for the next phrase; that is, the next call to GetPhrase will cause the text from that string to be returned. The string is appended to the command line and output to the command display stream. The string should not contain a terminating character.

If char is supplied, it is used as the terminating character and the next call to GetPhrase will return without giving the user any opportunity to edit the phrase. If char is omitted, the next GetPhrase will wait for the user to either type a terminating character (in which case the default phrase will be returned) or provide a replacement phrase followed by a terminating character.

BeginDefaultPhrase(cs)
Begins a default phrase. All occurrences of Puts(cs, char) between calls to

BeginDefaultPhrase and EndDefaultPhrase are included in the default phrase rather than treated as "noise" characters. This permits default phrases to be generated by arbitrary stream output.

EndDefaultPhrase(cs, char [])
Ends a default phrase started by BeginDefaultPhrase. If char is supplied, it is used as the terminating character, as described above under DefaultPhrase. BeginDefaultPhrase and EndDefaultPhrase must be paired and there must be no calls to GetPhrase between them.

2.3. Edit Control Procedures

These procedures control the operation of the command scanner in various ways. The procedures are passed as arguments to InitCmd, and some of them to GetPhrase. The default procedures are all defined in CmdScanDisplay.br, but the programmer is free to substitute other ones when appropriate.

The file CmdScanTty.br is an alternate to CmdScanDisplay.br, but oriented toward a minimal terminal stream interface. The only operations required are Gets and Resets on the keyboard stream and Puts on the display stream.

WordBreak(cs, char) = true|false
Returns true if char is a word break character and false otherwise. This controls the action of the control-W editing character and has no other effect. The default WordBreak procedure returns true only for space, escape, and carriage return.

PhraseTerminator(cs, char) = true|false
Returns true if char is a phrase terminating character and false otherwise. This controls the definition of a phrase, which is zero or more non-terminating characters followed by a terminating character. The default PhraseTerminator procedure returns true only for space, escape, and carriage return.

Echo(cs, char) = true|false
Returns true if char should be echoed when it is typed in and false otherwise. The default Echo procedure returns true if char is not a phrase terminator and false if it is (using whatever definition of phrase terminator is currently in effect).

Erase(cs, first, last, context)
Erases characters cs>>CS.buf>>Buf↑first through cs>>CS.buf>>Buf↑last (inclusive) from the output stream in whatever manner is appropriate for the medium. This interval may include both real phrase consituent characters and "noise" characters. Characters that were not echoed (i.e., not actually sent to the output stream) have #200 added to them and should be ignored.

The context argument indicates the context in which the Erase procedure is being called; this may be useful in determining the correct action.

eraseChar	A single character is being erased. (It is the character cs>>CS.buf>>Buf↑first; any other characters are "noise".)
eraseWord	A word (or phrase) is being erased.
eraseTerminator	The terminating character of the current phrase is being erased to permit additional editing on the phrase.

The default Erase procedure in the CmdScanDisplay module erases characters from the Alto display by means of EraseBits. If it is necessary to erase past the left margin (i.e., past a carriage return or a line wrap-around), the entire display window is erased and the command line is regenerated, thereby losing any text displayed before the beginning of the current command line. (This is necessary because the Operating System's display streams package generally does not permit one to manipulate other than the current display line.)

The default Erase procedure in the CmdScanTty module prints a backslash followed by the erased character in the eraseChar case and a left arrow in the eraseWord case.

Error(cs, ec)
This is the stream error procedure for cs and is called under a variety of exceptional conditions. The error codes (ec) are defined in CmdScan.decl. Most of them indicate a specific error condition. However, a few simply request a certain action and are therefore generally useful in client command parsing routines.

ecCmdDelete	(called from GetPhrase) The Delete character has been typed. The Error procedure should take appropriate action and should not return. The default Error procedure types "XXX" and forces a return from the call to InitCmd with value zero.
ecCmdTooLong	The command line buffer is full and an attempt has been made to append another character to it. The maximum length is the maxChars argument to InitCmd. If the Error procedure returns the excess character is thrown away. The default Error procedure blinks the display, resets the keyboard, and returns. (The CmdScanTty module outputs a bell to the display stream.)
ecTooManyPhrases	Attempt to put more than maxPhrases phrases into the command line (maxPhrases is passed to InitCmd). This is an unrecoverable error, and the default Error procedure calls SysErr.
ecEndOfPhrase	Attempt to read characters past the end of the current phrase (by Gets(cs)). If the Error procedure returns, the result value is returned by Gets. The default Error procedure calls SysErr.
ecKeyAmbiguous	(called from GetKeyword, described later) An ambiguous keyword has been typed in. The default Error procedure blinks the display, resets the keyboard, and sends control back to an earlier point of interpretation so as to permit the user to type in more characters.
ecBackupReplace	This and the following error codes are not associated with specific errors but simply request that a certain action be performed. This one requests that control be sent back to the beginning of the current phrase to permit typein of a replacement phrase.
ecBackupAppend	Requests that control be sent back to the current phrase to permit the user to append to or edit it.
ecCmdDestroy	Requests that control be sent back to the InitCmd that

	began this command, forcing it to return zero. This is the same as ecCmdDelete except that "XXX" is not typed.
other	Any error code not listed above is assumed to be some sort of syntax error arising from a higher-level command interpreter (such as the ones in the CmdScanAux module). The default Error procedure handles all of them in the same way: it displays a question mark, blinks the display, resets the keyboard, and sends control back to an earlier point of interpretation so as to permit the user to replace or modify the current phrase.

The following additional Alto display-specific procedures are defined in CmdScanDisplay.br:

CmdError(cs, string [])
If a string is supplied, outputs it to the command display stream. Then blinks the display window and issues a Resets operation on the command keyboard stream. (This procedure is also defined in the CmdScanTty module, but it outputs a bell to the display stream rather than blinking it.)

InvertWindow(ds)
Inverts the polarity of the display stream ds. That is, if it is now being displayed black on white, changes it to white on black or vice versa.

2.4. Backing Up and Catch Phrases

When it becomes necessary to edit a phrase that has already been parsed (i.e., passed to the client program via GetPhrase and Gets), it is necessary to back up the interpretation of the command line to an earlier point so as to permit the modified phrase to be reparsed. This situation arises in several cases: the user backspaces past the beginning of the current phrase or deletes the entire command, or a syntax error is detected and the current phrase or a previous phrase must be replaced or modified.

Rather than requiring GetPhrase and every higher-level procedure that calls GetPhrase to provide a failure indication (which the caller must then test after every call), the Command Scanner Package makes use of some devious control transfer primitives to back up control to an earlier point of interpretation, usually without the client program's being aware of it.

In the simplest case, control is sent all the way back to the call to InitCmd that created the Command State (cs). InitCmd returns again with the same cs as before, and the entire command line is reparsed by the client program. Each call to GetPhrase (up to the phrase that is being modified) returns a phrase saved away in the command state, just as if it had just been typed in. The effects of the command scanner procedures during a reparse are indistinguishable from those during the initial parse.

This control structure does have certain consequences that the programmer must be aware of. The first is that the context of the call to InitCmd must remain valid throughout the lifetime of the cs; that is, the procedure that called InitCmd must not return until the cs has been destroyed.

Second, the interpretation of a given command line must have constant effects. That is, the result of reparsing the command must be indistinguishable from the result of parsing it initially--there must be no incremental or time-dependent variations in interpretation.

There are situations in which resources are allocated during the course of command interpretation, e.g., storage blocks or open files. In some cases, when control is sent to an earlier point of interpretation, it is necessary to release such resources. The package provides a "catch phrase" mechanism by means of which the program can regain control so as to perform such cleanup. (The name is borrowed from Mesa, but the facility is not really very much like the Mesa "signal" and "catch phrase" machinery.)

The catch phrase mechanism is accessed through the following procedures:

EnableCatch(cs) = true|false

When this call is encountered during normal interpretation, EnableCatch saves away the current frame and pc in storage associated with the next phrase (the phrase that will be read by the next call to GetPhrase). In this context, EnableCatch always returns false.

While interpretation is being backed up, if a phrase is encountered for which an EnableCatch has been done, control is sent to that point; i.e., EnableCatch returns, but with value true rather than false. The programmer should write a statement of the form:

```
if EnableCatch(cs) then [ <catch phrase>; EndCatch(cs) ]
```

where <catch phrase> is code that performs the necessary cleanup.

EndCatch(cs)

Should be included at the end of every catch phrase. If control is being returned to a point of interpretation at or after the current phrase, EndCatch simply returns, thereby starting the reparse of succeeding phrases. However, if control is being sent back to a phrase before the current one, EndParse resumes the reverse transfer of control. Hence catch phrases are executed in reverse order, and the backing up of interpretation terminates at the latest catch phrase preceding the first phrase that must be reparsed.

DisableCatch(cs)

Undoes the effect of a previous EnableCatch for the current phrase. It may be issued before or after the GetPhrase that reads the current phrase. It is useful in situations where resources are allocated temporarily, across only one call to GetPhrase. The typical context is something like:

```
if EnableCatch(cs) then [ <release resources>; EndCatch(cs) ]
<allocate resources>
GetPhrase(cs)
<release resources>
DisableCatch(cs)
```

CmdErrorCode(cs) = ec

If control is being backed up due to an error (including a command delete), this procedure returns the error code. If the user backspaced past the beginning of a phrase, zero is returned. This procedure returns a valid result only in the context of a catch phrase.

As is the case for InitCmd, the context of every call to EnableCatch must remain valid during subsequent command interpretation. Effectively this means that calls to EnableCatch must be at the same or successively increasing depths of procedure calls.

Also, only one catch phrase may be enabled per phrase in the command line. The call to EnableCatch must precede the call to GetPhrase for the particular phrase, though it may either precede or follow a DefaultPhrase providing a default value for that phrase. This restriction makes inclusion of catch phrases within iterations somewhat tricky, though it is still possible.

A backup of interpretation is normally initiated only from within the Command Scanner Package itself, or from within an Error procedure called due to a syntax error. However, one may explicitly back up control by means of one of the following procedures:

BackupPhrase(cs, nPh [0], editControl [editReplace], char [])
Sends control backward nPh phrases relative to the current phrase (the default, zero, means restart interpretation of the current phrase). Note that BackupPhrase never returns. editControl determines the disposition of the current phrase, and may have one of the following values:

editNew	Discard the phrase and start over. (This option is not usually meaningful in the context of BackupPhrase, but is in ErasePhrase, described below.)
editAppend	Discard the phrase terminator and permit the user to append more characters to the phrase (or otherwise edit it).
editReplace	Discard the phrase terminator. If the first character typed by the user is a non-terminating, non-editing character, erase the entire phrase and start over (treating that character as the first character of the phrase); if it is an editing character, permit the user to edit the phrase as it stands; if it is a terminator, attempt to parse the phrase again with that terminator.

If char is provided, it is effectively inserted at the front of the command keyboard stream and is used the next time GetPhrase needs to input a character from the user.

ErasePhrase(cs, nPh [0], editControl [editReplace], char [])
Same as BackupPhrase, but first erases all intervening phrases (both from the command line buffer and from the display). In this case, the editControl parameter applies to the target phrase rather than to the current phrase. The target phrase is erased only if editControl is editNew.

3. Auxiliary Command Interpreters

The procedures in the CmdScanAux module each read a phrase (by calling GetPhrase) and interpret it in some way. While they are useful in their own right, they also serve as a good model for additional command interpretation procedures.

In general the procedures return only if successful and call Errors with an appropriate error code otherwise. As previously explained, the default handling for these errors consists of backing up control to the beginning of the current phrase and permitting the user to replace or modify the phrase. Also, these procedures interpret only the phrase itself, not the terminating character. It is the caller's reponsibility to check the terminator if required.

GetNumber(cs, radix [10]) = number
Returns the next phrase as a number in the specified radix. If an error occurs, Errors(cs, ec) is called with one of the following error codes:

ecEmptyNumber	The phrase is empty.

ecNonNumericChar	The phrase contains a character that is not a digit in the specified radix.
ecNumberOverflow	The number overflows 16 bits.

GetString(cs, PhraseTerminator [default], Help [], helpArg [], Echo [default]) = string
Returns the next phrase as a BCPL string. The optional arguments, if supplied, are passed to GetPhrase. The string is allocated from the same zone used to create cs.

GetFile(cs, ksType, itemSize, versionControl, hintFp, errRtn, zone, logInfo, disk) = stream
Calls OpenFile on the file whose name is the next phrase. All the arguments after cs are optional and are defaulted precisely as in OpenFile. If the file cannot be opened, calls Errors(cs, ecCantOpenFile).

Confirm(cs, string []) = true|false
Outputs the message "[Confirm]" preceded by the string if supplied. Then inputs a confirmation character and returns true if it is "Y" or carriage return and false if it is "N". Any other (non-editing) character causes Errors(cs, ecBadConfirmingChar) to be called. (Note that if Delete is typed, Confirm will not return but rather the entire command will be aborted.)

GetKeyword(cs, kt, returnOnFail [false], PhraseTerminator [default]) = entry
Looks up the next phrase in the keyword table kt (described later) and returns a pointer to the corresponding table entry. If the phrase is ambiguous, calls Errors(cs, ecKeyAmbiguous). If the phrase is not found, normally calls Errors(cs, ecKeyNotFound); however, if returnOnFail is true then returns zero in this case.

If a unique initial substring match occurs and the terminating character has not been echoed, appends the remainder of the matching keyword to the command line and to the display as if it had been typed in.

4. Keyword Package

This portion of the Command Scanner Package implements operations on an object called a Keyword Table. It is independent of the rest of the package and does not make use of any of its facilities. However, the CmdScanAux module does require the Keyword Package or some other package implementing equivalent operations.

The Keyword Package consists of two principal modules. File KeywordInit.br contains procedures to create and modify a keyword table, while Keyword.br contains procedures to look up keywords and to enumerate and destroy the table. The reason for this division is to permit one to create all needed keyword tables at program initialization time and then to discard the code (which accounts for more than half the total size of the package).

This package requires the StringUtil module of the Strings package, which in turn requires the ByteBlt package.

All keyword table operations except CreateKeywordTable are actually accessed through the Calls mechanism (Call0, Call1, etc.), so alternate implementations of the same interface are possible. In particular, the CmdScanAux module requires only that the LookupKeyword and EnumerateKeywordTable operations be provided.

A keyword table is an ordered set of <key, entry> pairs. The keys are BCPL strings and are maintained in alphabetical order for efficient lookup. The entries are fixed-length records whose interpretation is not defined by the package. While the lookup operation is efficient, the insert and delete operations are not, so this package is not suitable for maintaining large dictionaries or symbol tables. Its principal use is maintaining tables of keywords for applications such as command interpreters.

Procedures contained in the KeywordInit module are:

CreateKeywordTable(maxEntries, lenEntry [1], zone [sysZone]) = kt
    Creates and returns a keyword table (kt) capable of holding a maximum of maxEntries entries of lenEntry words each. The keyword table is allocated from the supplied zone and is initialized to empty.

InsertKeyword(kt, key) = entry
    Inserts the supplied key (a BCPL string) into the keyword table kt and returns a pointer to the corresponding entry, which is initialized to all zeroes. The key string is copied; storage for the copy is obtained from the zone passed to CreateKeywordTable. It is the caller's responsibility to appropriately initialize the contents of the entry. If the keyword table is full or a duplicate entry is inserted, SysErr is called.

DeleteKeyword(kt, key)
    Deletes the specified key (and its corresponding entry) from the keyword table kt. It is the caller's responsibility to dispose of any allocated objects pointed to by the deleted entry. If the key is not present in the table, SysErr is called.

Procedures contained in the Keyword module are:

LookupKeyword(kt, key, lvTableKey []) = entry
    Looks up the supplied key in the keyword table kt, returning a pointer to the corresponding entry if successful and zero if unsuccessful. For a successful lookup, the supplied key must either completely match a key in the table or be an initial substring of exactly one key. Upper- and lower-case letters are considered equivalent.

    If lvTableKey is supplied, a pointer to the full text of the matching keyword is stored in @lvTableKey if either a successful match or an ambiguous substring match occurs (zero is stored otherwise). In the case of an ambiguous substring match, the key stored is the first one that matches. This string is the one actually kept in the table (not a copy), so the caller must not modify it.

EnumerateKeywordTable(kt, Proc, arg)
    Calls Proc(entry, kt, key, arg) for each entry in the keyword table kt. The called procedure may modify the entry but must not insert or delete keys.

DestroyKeywordTable(kt)
    Destroys the keyword table kt, returning the table object and all keys to the zone from which they were allocated. It is the caller's responsibility to dispose of any allocated objects pointed to by entries in the table.

Additionally, the following procedure (defined in Keyword.br) may be of interest:

BinarySearch(key, tbl, lenTbl, Compare) = index
    Searches for key in the sorted table tbl, which has entries numbered zero to lenTbl-1 (inclusive). The comparison procedure Compare(key, tbl, i) is expected to compare key against entry i in the table and return a negative number if the key is "less than" the entry, zero if "equal", or a positive number if "greater than". All knowledge of the format of key and tbl is vested in the Compare procedure.

If the requested key is found, BinarySearch returns the index of the matching entry in the table. If the key is not found, -i-1 (= not i) is returned, where i is the index of the first entry greater than the requested one (i.e., the key before which the requested key should be inserted).

Bcpl Context Package

A tiny software package is available that provides facilities for managing multiple execution contexts for Bcpl procedures. A "context", as used here, is a region in which some part of a Bcpl stack is stored, including a "resume address" at which execution in the context can be resumed. Contexts may be strung together on "context lists." Such a list is "called" with CallContextList, which resumes the first context on the list until it "Block"s, then resumes the next context on the list, etc. Typically, each context that is resumed will execute a test to see if it really has work to do, and if not immediately Block again. Because running down the list resuming contexts is extremely rapid (the cost of switching between contexts is only 14 instructions), it is feasible to maintain rather large clouds of contexts in this way.

The package also includes an optional, very rudimentary time-slicing scheduler whose purpose is to reduce the frequency (and hence the cost) of context switches among "active" contexts.

The relevant files are contained in Context.dm. The basic context package consists of files Context.br, which contains about 50 instructions that must always be resident, and ContextInit.br, which contains initialization code that may be discarded after all contexts have been initialized. The optional time-slicing scheduler extension consists of ContextSched.br (resident, about 30 instructions), and ContextSchInit.br (initialization). The sources for these may be found in ContextSource.dm, which also includes a set of command files and Contextex.Bcpl, the example program given at the end of this writeup. A Nova version of this package is available.

1. Basic Context Package

ctx=InitializeContext(region, length, proc, extraSpace [0])

This procedure initializes a context, using a block of storage starting at address "region," of length "length" for the stack and sundry other information. The "proc" argument specifies a procedure to call the first time the context is resumed. The optional parameter "extraSpace" allows the context to contain other information of the user's choosing.

The result of the procedure is a CTX structure:

```
structure CTX:
          [
          Next      word      //Pointer to successor context
          Stack     word      //Current stack pointer
          StackMin word       //Stack limit
          user      word extraSpace //For user's purposes
          stackArea word remaining //The stack area
          ]
```

The caller is expected to build context lists by chaining through the Next entries. InitializeContext sets Next to zero. Note that this way of managing context lists is consistent with the conventions used in the Alto Queue package.

The "caller's frame" pointer in the first frame of the context is initialized to zero. This enables programs that enumerate stacks (e.g., the Overlay package) to know when to stop.

CallContextList(ctx)

This function resumes each context on the list headed by ctx linked through CTX.Next entries. Each context executes until it calls the procedure Block. When the list is exhausted (a zero Next value terminates the list), CallContextList returns. CallContextList will never return if the list is linked into a ring.

The first time a context is encountered by CallContextList, the procedure given by the "proc" argument of InitializeContext is called, with the context itself as its argument. Any other parameters required to distinguish instances of contexts may be passed as an "extraSpace" block, which begins at ctx!3.

CallContextList is reentrant, and may be called from within an interrupt. This permits one to have hierarchies of contexts (with preemptive priority) simply by running all contexts of a given priority at an appropriate interrupt level (note that the interrupt necessary to cause execution of such contexts may be either hardware- or software-initiated). This is accomplished most conveniently by means of the Bcpl Interrupt Interface, described separately. Note that contexts running at different priority levels must protect common data bases and critical sections, whereas contexts at the same level are free from race conditions so long as they don't call Block from within critical sections.

Block()

Ceases execution of the calling context. Execution resumes the next time the context is encountered on some list by CallContextList.

If Block is called outside of any context (that is, no call of CallContextList is currently in progress), it returns immediately.

For debugging purposes, two statics defined in Context.br are of interest: CtxRunning contains the address of the context currently running, and CtxCaller points to the frame for the current invocation of CallContextList.

2. Time-Slicing Scheduler Extension

While the cost of switching between contexts is very small, in a system with many contexts the effective cost of a call to Block may be quite large due to the sheer number of other contexts that are resumed before control returns from this call to Block. Typically, most contexts are "waiting" rather than "active"; i.e., they are calling Block from within a loop that is waiting for some "wakeup" condition to occur. On the other hand, there are often one or two "active" processes that are performing some useful, long-running computation. For proper operation of the context package, it is necessary that such processes give up control reasonably often. But it is clearly wasteful to do so too often.

This extension to the basic context package introduces a new primitive called Yield, which is similar to Block except that it does not always actually give up control (i.e., sometimes it just returns immediately). Specifically, if the present context has been executing for less than one time slice, Yield returns immediately. In this implementation, the time slice is between 17 and 34 milliseconds.

Thus, Block and Yield are both procedures for relinquishing control, but with slightly different interpretations. Block should be called from within wait loops, whereas Yield should be called from within code that is doing "useful" computation. In the latter case, if the present context's time slice has not expired, Yield returns immediately after executing only three instructions.

The time-slicing scheduler must be initialized by calling InitContextSched(), whose code may subsequently be discarded. Yield behaves the same as Block until this initialization has been performed.

3. Example

The following trivial program initially establishes two contexts and chains them together into one list. One context (running CommandProc) simply blocks until something is typed on the keyboard, then treats the typein as a command. The second waits for an Ethernet message to arrive, and types out "Message arrived."

When the letter "S" is typed to CommandProc, a new context is created to run TimerProc. Each instance of a TimerProc context has associated with it an identifying integer N (stored in the extraSpace word Ctx!3) which it prints out at intervals of N seconds.

```
external [ InitializeContext; CallContextList; Block
           SerialNumber; Ws; Wns; Gets; Endofs; keys; dsp
           InitializeZone; Allocate]
manifest RTC=#430
manifest EPLoc=#600
manifest EICLoc=#604
manifest EIPLoc=#605
manifest ESLoc=#610
manifest SIO=#61004

static [ CtxZn; CtxHead; NumTimeProcs=0 ]


let main() be
           [
           let z=vec 10000; CtxZn=z  // Zone to allocate contexts from
           InitializeZone(CtxZn,10000)
           let s1=vec 200
           let s2=vec 200

           CtxHead=InitializeContext(s1, 200, CommandProc)
           let next=InitializeContext(s2, 200, EtherProc)
           @CtxHead=next

           CallContextList(CtxHead) repeat
           ]


and CommandProc() be
[
Ws("*n**")
while Endofs(keys) do Block()  // Block until user types something
let Char=Gets(keys)
switchon Char into
           [
           case $S: case $s:
                      [
                      Ws("*nStart another TimeProc")
                      let region=Allocate(CtxZn,200)  // Create new context
```

```
                    let ctx=InitializeContext(region,200,TimeProc,1)
                    NumTimeProcs=NumTimeProcs+1
                    ctx!3=NumTimeProcs  // Parameter for this instance
                    ctx!0=CtxHead; CtxHead=ctx  // Link into context list
                    endcase
                    ]
          case $Q: case $q:  [ Ws("Quit"); finish]
          default: Ws("?")
          ]
] repeat

and TimeProc(Ctx) be
          [
          let interval=Ctx!3        // Get interval from context
          let f=@RTC+27*interval   // That many seconds from now
          until (@RTC-f) gr 0 do Block()
          Wns(dsp,interval)         // Type our interval
          ] repeat

and EtherProc() be
          [
          StartIO(3)                //Reset Ether
          @ESLoc=SerialNumber
          let buf=vec 50
          @EICLoc=50
          @EIPLoc=buf
          @EPLoc=0
          StartIO(2)                //Start input
          until @EPLoc ne 0 do Block()
          if (@EPLoc rshift 8) eq 0 then Ws("Message arrived")
          ] repeat

and StartIO(c) be (table [ SIO; #1401 ])(c)
```

4. Revision History

November 17, 1976: Calling Block() when not in a context is now a no-op rather than giving rise to weird crashes; InitializeContext sets the first frame's "caller's frame" pointer to zero.

May 21, 1977: Time-slicing extension added; CallContextList speeded up.

Diablo Printer Package

This package provides a standard stream interface to the Diablo Printer. The facilities provided are limited to simulation of a conventional Ascii terminal using a fixed-pitch font. The software is derived from a version of the Diablo primitives used in Bravo, courtesy of Greg Kusnick.

The package consists of a single binary file, DiabloPrinter.br. The source for this, DiabloPrinter.bcpl, is included in DiabloPrinter.dm, which also contains a test program, DiabloType.bcpl, which types an arbitrary text file on the Diablo printer.

Besides using standard operating system facilities, this package makes use of the Context and Timer packages. If one desires not to include the Context package, it suffices to define an external procedure Block() that returns immediately.

There is only one externally-callable procedure, which works as follows:

CreateDiabloStream(charWidth [6], charHeight [8], pageWidth [450], pageHeight [528], leftMargin [0], zone [sysZone]) = dps

Creates a Diablo Printer Stream (dps) using the supplied parameters, all of which are optional. Width and height arguments are in units of 1/60 and 1/48 inch respectively, and cannot be greater than 1023. charWidth and charHeight define the width and height of each character, including inter-character and inter-line spacing. The defaults are appropriate for standard typewheels such as Elite 12. pageWidth and pageHeight define the printing area on each page. The defaults are appropriate for 7.5 inches wide (assuming half-inch margins) by 11 inches high (no margins). With the standard font size, this permits 75 characters per line and 66 lines per page. leftMargin specifies the position of the logical left margin relative to the extreme left limit of the carriage (note that leftMargin is not included in pageWidth). The zone argument specifies the zone to be used to allocate the stream structure.

The following operations are defined on a Diablo Printer Stream:

Puts(dps, char)

Prints the specified character. All printing characters (Ascii codes 40-177) are typed with whatever is in the corresponding position on the typewheel, with the exception of "←" which is printed by overstriking "-" and "<" (since typewheels tend to have the underline character in this position).

The following non-printing characters (Ascii 0-37) are interpreted to provide the specified functions. All other non-printing characters are ignored.

15 (return)	Returns the carriage to the logical left margin and advances the paper to the next line.
11 (tab)	Positions the carriage to the next multiple of 8 character positions.
10 (backspace)	Backs up the carriage by one character position (ignored if already at the logical left margin).
14 (form feed)	Advances the paper to the beginning of the next page. (The beginning of the first page is defined by where the paper was positioned when CreateDiabloStream was called).

If the right margin is exceeded, an automatic carriage return is executed.

If a hardware problem is detected, Errors(dps, code) is called, where code is ecDiabloPrinterNotReady if an operation did not complete within a reasonable time (one second) and ecDiabloPrinterCheck if the printer reported a "check" error. The default Errors procedure is SysErr. If the Errors procedure returns, the operation is retried. Note that the printer must be reset in order to proceed after a "check" error (see below).

Stateofs(dps) = true or false
Returns true if the hardware is reporting that it is ready to execute a new operation. Note that this is not a guarantee that an attempt to print a character will succeed, since printing a character generally involves several successive operations.

Resets(dps)
Resets the printer hardware and restores the carriage to the physical left margin. This operation must be performed to recover from a "check" error.

Closes(dps)
Destroys the stream. This includes returning the stream structure to the zone from which it was allocated.

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.

Pup EFTP Package

The routines described here implement the EFTP protocol, a simple ack-per-packet protocol built on level 1 of Pup. The EFTP protocol is used by the EFTP subsystem to send files among machines, by Bravo and Gears to send files to Ears for printing, and by gateways to send boot files and update internal data bases. It is a place-holder for the Reliable Packet Protocol which will be implemented in the next iteration of Pup.

The EFTP protocol is documented in <pup>EFTPSpec.ears. The source file for these routines, PupEFTP.bcpl, and its declaration file, PupEFTP.decl are contained in <altosource>EFTP.dm (along with the sources for the EFTP subsystem). Get a copy of these two files and look at the code while reading the description that follows. This documentation assumes you are familiar with the Pup package, and its supporting environment. All timeouts are in units of 10 ms.; a timeout of -1 means infinity.

1. The Routines

InitEFTPPackage(zone)
This procedure is currently a no-op, but may be used in the future, should it become necessary to initialize and allocate free storage within the package.

OpenEFTPSoc(soc, lclPort [defaulted], frnPort [zeros])
Opens a Pup level 1 socket and creates an EFTPSoc. "soc" should point to a block of size lenEFTPSoc.

CloseEFTPSoc(soc)
Releases any PBIs held in the EFTP part of soc, and then closes the Pup level 1 socket.

SendEFTPBlock(soc, addr, count, timeout) = byte count or error code
Constructs an EFTP data packet from the information in soc, copys count bytes beginning at addr into the data part of the Pup, and transmits it. This routine manages retransmissions, returning count if the packet is acknowledged within the timeout, or a negative error code if some abnormal condition occured.

ReceiveEFTPBlock(soc, addr, timeout) = byte count or error code
Copys the data from the next in-sequence data packet into memory beginning at addr and returns the number of bytes received (532 max), or a negative error code if some abnormal condition occured. If timeout is -1, ReceiveEFTPBlock will wait indefinitely until the next packet is available, otherwise it will return an error if no packet becomes available within the timcout. If the next in-sequence packet is an EFTP End, this routine will perform the end sequence and return a byte count of zero.

SendEFTPEnd(soc,timeout) = true/false
Initiates an end sequence with the EFTP receiver, managing retransmissions, and returns true if the sequence is completed correctly within the timeout.

GetEFTPAbort(soc) = PBI
Returns a pointer to the most recently received EFTPAbort, should the user want to look at it. If no abort has been recieved, zero is returned.

SendEFTPAbort(soc, abortCode, abortString)
Builds and transmits an EFTP Abort packet with abortCode and abortString as data.

2. Error Codes

EFTPTimeout = -1
The requested operation did not complete within the timeout specified in the call. Returned by SendEFTPBlock, ReceiveEFTPBlock, and SendEFTPEnd.

EFTPAbortReceived = -2
An EFTP Abort was received while performing the requested operation. GetEFTPAbort(soc) will return a pointer to the abort packet. Returned by SendEFTPBlock and ReceiveEFTPBlock.

EFTPAbortSent = -3
A serious protocol violation was noticed while performing the requested operation. There is no hope of continuing. An EFTP Abort was sent to the other end. Returned by SendEFTPBlock, ReceiveEFTPBlock, and SendEFTPEnd.

EFTPResetReceived = -4
While waiting for the next in-sequence data packet in an ongoing transfer, a data packet with sequence number zero was received from the other end. Returned by ReceiveEFTPBlock.

Alto Ethernet Boot Package

The EtherBoot package (file EtherBoot.br) consists of an Alto Ethernet boot loader and a small amount of additional code enabling a program to terminate execution of itself and boot-load a new program from the Ethernet.

EtherBoot(bfn)
: Copies a small (256 word) Ethernet boot loader into low memory and transfers control to it with 'bfn' (boot file number) as an argument. The loader begins broadcasting "Mayday" messages with bfn as data, on the local Ethernet. A server that hears this message and has a copy of the boot file matching bfn will connect to the Alto and send the file by means of the EFTP protocol.

The boot loader contained in this package is identical to the one invoked when the Alto's boot button is pressed with the <bs> key and zero or more other keys down. The correspondence between bfn's, boot files, and key combinations is given below. However, note that calling EtherBoot differs from actually booting the Alto in one way: tasks are not reinitialized to run in the Rom, since no hardware reset is actually performed.

Mayday servers will keep copies of some useful programs in boot format (see BuildBoot.tty for how to create a bootable file). An obvious application would be for the Executive to boot DMT from Ethernet when the disk is turned off. Boot files presently kept by Mayday servers are:

bfn	file	equivalent boot key combination	
		Alto-I	Alto-II
0	DMT.boot	bs	bs
1	SYS.boot	bs blank-top	bs bw
2	FTP.boot	bs blank-middle	bs fr4
3	Scavenger.boot	bs blank-top blank-middle	bs bw fr4
4	CopyDisk.boot	bs]	bs]

Event Report

The EventReport package provides a convenient interface to the Pup Event Report protocol (see relevant Pup documentation elsewhere for details). This protocol is used for logging errors of various kinds (e.g., parity errors) and for keeping records of resource utilization (e.g., number of pages in a printer run).

EventReport(eventV, eventVLength[0], eventPort[ErrorLogAddress], retryCount[3], timeOut[3*27])

This subroutine reports an event recorded in the vector eventV. The remaining arguments (with defaults shown in brackets are): eventVLength, the number of words in the event recorded in eventV; eventPort, a pointer to a Port (Pup terminology and format) to which the event should be sent; retryCount, the number of times the transmission will be attempted; and timeOut, the time to await a response from each retry before giving up (in units of 1/27 second).

EventReport returns "true" if the event was successfully logged, or "false" if it was unable to log the event (perhaps because the Alto has no Ethernet).

FindPkg - a fast file searching package

This package uses the Alto microinstruction RAM to search standard Alto files for certain simple kinds of patterns at very high speed (it normally keeps up with the disk). It is written in Bcpl and uses the ScanFile package for doing the actual disk transfers.

To use the package, one first "compiles" the pattern into specialized microcode which is loaded into the RAM, and then scans as many files as desired using this microcode. To compile the pattern, call

FindCompile(pattern, chartab[, wildchar, fuzz, outstream, storeproc, regtable])

where all the arguments beyond chartab are optional. The arguments have the following significance.

Pattern is a Bcpl string, the pattern being searched for. The search ignores the high-order bit of characters in both the file and the pattern. In addition, the following 3 arguments affect how the pattern is interpreted. The maximum length of the pattern is the number of R and S registers available (see below), rounded down to an even number if necessary.

Chartab is a 200b-word array which specifies how characters in the file are to be interpreted. Chartab!j specifies how occurrences of the character whose code is j are to be treated. The possible contents of each chartab entry are: classSkip, meaning ignore the character completely; classOther, meaning that the character is to be taken literally; or a code between 0 and 177b inclusive, meaning that the character is to be treated as though it were that character (which, in turn, must be of classOther in the table). For example, to cause lower case letters in the file to be treated as though they were the corresponding upper case letter, set chartab!$a = $A, etc.

Wildchar is a character whose appearance in the pattern string means "match any character in the file". For example, if the pattern string is "A?B" and wildchar is $?, any occurrence of A followed by any character followed by B in the file will be considered an occurrence of the pattern. If wildchar is not a character code, it is ignored, and all characters in the pattern are taken literally. Wildchar defaults to -1 (take the pattern literally).

Fuzz is the number of mismatches between the pattern and the corresponding string in the file that will be tolerated. For example, if the pattern is ABCD, then with fuzz=0, only the string ABCD in the file (after interpretation through chartab) will match; with fuzz=1, the strings ABCX, ABXD, AZCD, or ZBCD would match, and so on. Note that fuzz only applies to replacement mismatches, not insertions (e.g. ABXCD), deletions (e.g. ABD), or transpositions (e.g. ABDC). Fuzz defaults to 0 (exact match required).

Outstream, if non-zero, is a character stream on which FindCompile will write a listing of the microcode it generates. This is only useful for debugging. Outstream defaults to 0 (no listing).

Storeproc determines what will be done with the microcode. Storeproc=false means discard it (although a listing will still be produced if outstream is non-zero). Storeproc=true means store it in the RAM for execution. Otherwise, FindCompile calls storeproc(location, insvec) for each instruction it generates, where insvec is a 2-word vector containing the microinstruction. Storeproc defaults to true (store for execution).

Regtable is a 4-word bit table that specifies what R and S registers are available for use by the microcode. These registers must not be used by other

tasks, or by the Nova instruction set, although they may be used by BitBlt or other Alto-specific instructions. Also, registers 14b through 16b are assumed usable, and should not appear in the bit table. Regtable defaults to a table that lets the microcode use register 17b and registers 41b through 76b, which will accommodate a 30-character pattern.
FindCompile normally returns zero. If it encounters any difficulties, it returns a string which describes the difficulty. This string is meant to be printed for the user, not interpreted by the calling program.

After calling FindCompile to load the RAM, one scans files as follows. First, call ScanFile to initialize the scan (see the documentation for ScanFile for how to do this). ScanFile returns an object called a sfd (ScanFile descriptor). To start searching the file, call

```
        FindInit(sfd, fa)
```

where sfd is the value from ScanFile and fa is a file address (FA) structure into which FindPkg will store each time it finds a match. Then to find each match in turn, call

```
        FindNext()
```

FindNext either finds the next match or scans to the end of the file. In the former case, it returns a non-negative number that says how many characters of the pattern had been examined before it decided it had a match, and stores the disk address, page number, and character position at that time into the fa given to FindInit. For example, if the pattern is "ABCD" and fuzz=1, then if the file contains ABXD, FindNext will stop after the D and return 4, while if it contains ABCX, it will stop after the C and return 3, since it knows it has a match at that point regardless of the next character. If FindNext runs off the end of the file, it returns -n-1 where n is the number of pages in the file. The calling program should then call ScanFinish(sfd) to clean up the ScanFile data structures.

FindPkg consists of 3 files:
FindNext.BR, containing the procedures FindInit and FindNext;
FindCompile.BR, containing the procedure FindCompile;
FindPkgDefs.D, a Bcpl source file containing the definitions for the character classes.

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. FTR applied to an accumulator containing 1.5 is 1; to one containing -1.5 is -1. 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
```

FEXP(acnum,increment)	Adds 'increment' to the exponent of the specified accumulator. The exponent is a binary power. Thus FTR(FEXP(FLDI(1,1),4))=16.
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)	Truncates the contents of the floating point ac and stores it 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. Same conventions for integer and fractional part as FSTDP.
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. Saving and Restoring Work Area

FLOAT has a compiled-in work area for storing contents of floating accumulators, etc. The static FPwork points to this area. The first word of the area (i.e. FPwork!0) is its length and the second word is the number of floating point accumulators provided in the area. The routines use whatever pointer is currently in FPwork for the storage area. Thus, the accumulators may be "saved" and "restored" simply by:

```
let old=FPwork
let new=vec enough; new!1=old!1 //Copy AC count
FPwork=new
...routines use "new" work area; will not affect "old"
FPwork=old
```

This mechanism also lets you set up your own area, with any number of accumulators. The length of work area required is 4*(number of accumulators)+constant. (The constant may change when bugs are fixed in the floating point routines. As a result, you should calculate it from the compiled-in work area as follows: constant←FPwork!0-4*FPwork!1.) It is not essential that the length word (FPwork!0) be exact for the routines to work.

5. 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

The result of the error routine is returned as the result of the offending call to the floating point package.

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.

Pup File Transfer Protocol Package

This package is a collection of modules implementing the Pup File Transfer Protocol. The package is used by the FTP subsystem and the Interim File System and runs on Altos and Novas.

1. Overview

This document is organized as a general overview followed by descriptions of each of the modules in the package. A history of revisions to the package is included at the end.

Before beginning the main documentation, some general comments are in order.

a. The File Transfer Protocol is (alas) complex; this package requires the Pup package and all of its supporting packages plus some other packages not specific to Pup. This documentation is less tutorial than normal Alto package descriptions so you should be prepared to consult its author.

b. This document describes the external program interfaces for a particular implementation of the File Transfer Protocol, and does not deal with the internal implementation nor the reasons for design choices in the protocol or this implementation. Before considering the details of this package, you should read <Pup>FtpSpec.ears to get the flavor of how the File Transfer Protocol works. The <Pup> directory also contains descriptions of the lower level protocols on which FTP is based. Detailed knowledge of these protocols is not necessary to use this package, but you must be familiar with the operation of the Pup package.

c. This package and the protocol are under active development so users should expect modifications and extensions.

d. This package is designed to run on both Altos and Novas, under several operating systems and with several file systems. Functions are carefuly split into protocol-specific and environment-specific modules. This package provides the protocol modules; you must write the matching environment-specific modules.

1.1. Organization

The FTP package comes in four modules: Server, User, Utilities, and Property lists. The utility and property list modules are shared by the User and Server.

The User and Server modules implement their respective halves of the protocol exchanges.

The Property List module generates and parses property lists, filesystem-independent descriptions of files. When passed between User and Server FTPs through the network byte stream, their form is defined by protocol as a parenthesized list. When passed between these protocol modules and the user-supplied modules in a program, they take the form of a data structure defined by this package.

The Utility module contains protocol routines shared by the User and Server modules and some efficient routines for transferring data between a network stream and a disk stream.

1.2. File Conventions

The FTP package is distributed as file FTPPackage.dm, and contains the following files:

User	
FtpUserProt.br	User protocol common to file and mail
FtpUserProtFile.br	User file commands
FtpUserProtMail.br	User mail commands
Server	
FtpServProtFile.br	Server file commands
FtpServProtMail.br	Server mail commands
Property lists	
FtpPListProt.br	Property list protocol
FtpPList1.br	Implements a 'standard' property list
FtpPListInit.br	Initialization
Utility	
FtpUtilB.br	Common protocol
FtpUtilXfer.br	Unformatted data transfer
FtpUtilDmpLd.br	Dump/Load data transfer
FtpUtilA.br	Assembly-language utility code
FtpUtilInit.br	Initialization
Definitions	
FtpProt.decl	Protocol parameters and structures
Command files	
CompileFtpPackage.cm	Compiles all files
DumpFtpPackage.cm	A list of all binary files
FtpPackage.cm	A list of all source files

All of these modules are swappable, and are broken up into pieces no larger than 1024 words. Modules whose names end in "init" are initialization code which should be executed once and thrown away.

The source files are kept with the subsystem sources in FTP.dm and are formatted for printing in a small fixed-pitch font such as Gacha8 (use the command 'Gears/s @FtpPackage.cm@').

1.3. Other Packages

FTP is a level 3 Pup protocol, and this package uses a number of other Alto software packages. As always, files whose names end in "init" may be discarded after initialization (except ContextInit.br).

Pup Package			
PupBSPBlock.br	PupBSPStreams.br	PupBSPProt.br	PupBSPa.br
PupRTP.br	PupDummyGate.br	PupRoute.br	
Pup1b.br	PupAl1a.br	Pup1Init.br	
PupAlEthb.br	PupAlEtha.br	PupAlEthInit.br	
Context Package			
Context.br	ContextInit.br		
Interrupt Package			
Interrupt.br	InterruptInit.br		
Queue Package			
AltoQueue.br			
Timer Package			
AltoTimer.br			
Time Package			
CTime.br			
ByteBLT Package			
AltoByteBLT.br			
CmdScan Package			
Keyword.br	KeywordInit.br		
Strings Package			
StringUtil.br			
Template Package			
Template.br			

1.4. Principal Data Structures

The following data structures are of interest to users, and together with the procedures described later, constitute the package interface.

FPL	File Property List, is this implementation's internal representation of the protocol-specified property list. An FPL structure will be referred to as a 'pList' from here on.
FTPI	File Transfer Package Interface, contains pointers to the network byte stream, user disk stream, log stream, the file buffer, and various flags.
FTPSFI	FTP Server File Interface, is a vector of user-supplied procedures constituting the interface between the protocol and environment-specific modules in a file Server.
FTPSMI	FTP Server Mail Interface, same as an FTPSFI except for a mail server.
FtpCtx	FTP Context, is the process-global storage for a User or Server FTP process. It consists of an FTPI, and if the process is a Server, an FTPSFI or FTPSMI. This is a convenient place for the user-supplied modules to keep process-private data. You can do this by adding items to the FtpCtx definition and then recompiling everything.

The entire FtpCtx need not be filled in all of the time. For each group of procedures, the items they require will be specified. A general description of the contents of the FTPI part of an FtpCtx is in order here.

bspSoc	a pointer to a BSP socket open to a remote FTP process.
bspStream	a pointer to the stream in the above BSP socket. Pup package experts will recognize that this is redundant, but it is often convenient and makes the code clearer.

dspStream	a pointer to a stream to which this package will output generally useful information, including copious amounts of debugging information if debugFlag is true. The only operation that need be defined is 'Puts'.
diskStream	a pointer to a disk stream. It should always be opened in byte mode.
buffer	a pointer to a block of memory which can be used for block transfer I/O operations. The bigger this is the faster things will go.
bufferLength	the length in words of the above buffer
debugFlag	a boolean. If true, the protocol exchanges for this context are output to dspStream as text, along with some other useful information. Use this! It will save you much head-scratching.
connFlag	a boolean. This should be true if bspSoc is open. The package will cooperate in maintaining this flag, which is valuble when finishing.
serverFlag	a boolean. This flag is tested by procedures in the shared modules to determine whether the caller is a User or Server.
savedBSPErrors	the default BSP error procedure is saved here. This package handles certain errors itself.

1.5. Programming Conventions

This package can be used with the Bcpl Overlay package. File FtpOEPInit.br contains a procedure which will help do this, but you should consult with the author.

This package does a lot of string manipulation, and uses the following conventions:

a. All strings are allocated from 'sysZone'.

b. Strings are represented in data structures (such as property lists) as addresses. Zero means no string is present.

All of the procedures in this package expect to execute in contexts (in the sense of the Context package), and expect CtxRunning (defined by the Context package) to point to an appropriately filled in FtpCtx.

1.6. Timeouts

If a Get or Put operation times out, the bspStream Get and Put routines are changed so that all subsequent operations fail immediately. This will cause the current command to fail quickly, so that its caller can take appropriate action. This package makes timeouts look the same as if the stream closed, and treats them as unretryable. Two timeouts are used by the package and kept in statics.

getCmdTimeout
This timeout is used in situations involving human user interaction and should be fairly long. Its default value is defGetCmdTimeout, defined in FtpProt.decl.

getPutTimeout

This timeout is used when transferring data and should be fairly short. Its default value is defGetPutTimeout, defined in FtpProt.decl.

2. Server

The FTP Server module consists of two files: FtpServProtFile.br, a file server, and FtpServProtMail.br, a mail server. The internal organization of both files is the same; they just implement different sets of commands. Each file has one external procedure:

FtpServProtFile() or FtpServProtMail()
which carry out protocol commands received over bspStream by calling the user-supplied procedures in FTPSFI or FTPSMI. When the BSP connection is closed by the remote FTP User process, these procedures return.

This module uses the following fields in FtpCtx: dspStream, bspStream, bspSoc, and FTPSFI or FTPSMI. All of the primary command slots (Version, Store, Retrieve, StoreMail, etc.) must contain procedures. If you do not wish to implement a command, it suffices to point the command's slot at the following procedure:

```
and NYI(nil) = valof
    [
    FTPM(markNo,1,"Unimplemented Command")
    resultis false
    ]
```

in which case any subsidiary procedures for that command (such as StoreFile and StoreCleanup for the Store command) need not be filled in. FTPM is described in more detail below. For the remainder of this section, 'FtpServProt' refers to 'FtpServProtFile' or 'FtpServProtMail'.

2.1. Version Command

By convention, Version is the first command exchanged over a newly opened FTP connection. The User sends its protocol version number and a string such as "Maxc Pup Ftp User 1.04 19-Mar-77". When FtpServProt receives this command, it replys with its protocol version number and then calls

```
(CtxRunning>>FtpCtx.Version)()
```

which should generate some herald text:

```
Wss(CtxRunning>>FtpCtx.bspStream, "Alto Pup FTP Server ")
```

to which FtpServProt will append a string of the form "1.13 14-May-77".

2.2. Retrieve Command

When the remote FTP User process sends the command 'Retrieve' and a property list describing the files it wants to retrieve, FtpServProt parses the property list and calls

```
(CtxRunning>>FtpCtx.Retrieve)(remotePList,localPList)
```

which should decide whether to accept the command. Retrieve's decision may involve checking passwords, looking up files, and other actions using the information in remotePList plus other environment-specific information, such as whether the requester has the correct capabilities, etc. To refuse the request, Retrieve should call

```
FTPM(markNo, code, string)
```

and return false. To accept the command, it should return a new pList describing a file matching remotePList which Retrieve is willing to send. FtpServProt will return this pList as 'localPList' in the next call to Retrieve, so that it can be deallocted. On the first call, localPList will be zero. Some FTP implementations require a minimum set of properties here, but the whole subject of who should specify what properties is rather involved and beyond the scope of this description. For more information, consult the FTP specification. This package provides a fast procedure (in the Utility module) for deciding the 'type' of a file (text or binary) which you may find useful.

Property lists in retrieve requests may specify multiple files, so FtpServProt will continue to call Retrieve until it returns false. On each call, remotePList will be the same original pList sent from the remote User, and localPList will be the last pList returned by Retrieve. If Retrieve supports multiple file requests then it must save some information so that the next time FtpServProt calls it, it can generate the next file. If Retrieve does not support multiple file requests then it should do its thing during the first call and remember that it is finished. The next time it is called it should return false having only deallocated localPList (it should not call FTPM).

If Retrieve returns true, FtpServProt sends the returned property list back to the User to more fully describe the file. At this point the User may back out of the transfer, in which case the next procedure will be skipped, and RetrieveCleanup will be called immediately. If the User indicates a willingness to proceed, FtpServProt then calls

```
(CtxRunning>>FtpCtx.RetrieveFile)(pList)
```

to transfer the file data. This package provides a procedure (in the Utility module) for transferring data from a disk Stream to a BSP Stream, but you are free write your own. When RetrieveFile has finished the transfer, it should return true if everything went OK.

Next, FtpServProt calls

```
(CtxRunning>>FtpCtx.RetrieveCleanup)(pList,ok)
```

where 'ok' is false if RetrieveFile returned false or the User backed out of the command. Note that if Retrieve returned true, RetrieveCleanup will always be called, but RetrieveFile may not. If Retrieve allocates any resources (such as opening a file) they should be deallocated here.

Finally, FtpServProt calls Retrieve again, and the process repeats until Retrieve returns false.

2.3. Store Command

When the remote FTP User process sends the command 'Store' followed by a property list describing the file, FtpServProt parses the property list and calls

```
(CtxRunning>>FtpCtx.Store)(pList)
```

which should decide whether to accept the command. To accept, Store need only return true; no property list is sent back in this command. To refuse the command Store should call FTPM(markNo, code, string) and return false, in which case the next procedure (StoreFile) is not called.

If Store returns true, FtpServProt tells the User process to go ahead and send the file, and then calls

```
(CtxRunning>>FtpCtx.StoreFile)(pList)
```

to transfer the file data. This package provides a procedure (in the Utility module) for transferring data from a BSP Stream to a disk Stream, but you may write your own. When StoreFile has finished the transfer, it should return true if everything went OK.

Finally, FtpServProt calls

```
(CtxRunning>>FtpCtx.StoreCleanup)(pList,ok)
```

where 'ok' is true if StoreFile returned true and the User indicated that everything went ok. If 'ok' is false, StoreCleanup should delete the file, since it is almost certainly damaged. Note that if Store returned true, StoreCleanup will always be called, but StoreFile may not. If Store allocates any resources (such as opening a file) they should be deallocated here.

2.4. Delete Command

When the remote FTP User process sends the command 'Delete' followed by a property list describing the files which it wants to delete, FtpServProt parses the property list and calls

```
(CtxRunning>>FtpCtx.Delete)(remotePList,localPList)
```

which should decide whether to accept the command. Don't delete anything yet! The User may still back out. To refuse the delete request, Delete should call FTPM(markNo, code, string) and return false. To accept the command, it should return a new pList with every property it can find, so that the User can be sure of the identity of file to be deleted, and return true. FtpServProt will return this pList as 'localPList' in the next call to Delete, so that it can be deallocted.

Property lists in delete requests may specify multiple files, so FtpServProt will continue to call Delete until it returns false. On each call, remotePList will be the same original pList sent from the remote User, and localPList will be the last pList returned by Delete. If Delete supports multiple file requests then it must save some information so that the next time FtpServProt calls it, it can generate the pList for the next file. If Delete does not support multiple file requests then it should do its thing during the first call and remember that it is finished. The next time it is called it should return false having only deallocated localPList (it should not call FTPM).

If Delete returns a Plist, FtpServProt will send it back to the User and wait for confirmation. If the User still wants to delete the file, FtpServProt calls

```
(CtxRunning>>FtpCtx.DeleteFile)(pList)
```

which should delete the file. Finally, FtpServProtFile calls Delete again, and the process repeats until Delete returns false.

2.5. Directory Command

When the remote FTP User process sends the command 'Directory' followed by a property list naming the files about which it wants information, FtpServProt parses the property lists and calls

(CtxRunning>>FtpCtx.Directory)(pList)

which should decide whether to accept the command. To refuse the request (because for example the requestor does not have the correct access capabilities) Directory should call FTPM(markNo, code, string) and return false. To accept the command it should return a pList describing a file.

Property lists in directory requests may specify multiple files, so FtpServProt will continue to call Directory until it returns false. If Directory supports multiple file requests then it must save some information so that the next time FtpServProt calls it, it can generate the pList for the next file. If Directory does not support multiple file requests then it should do its thing during the first call and remember that it is finished. The next time it is called it should return false having only deallocated localPList (it should not call FTPM).

2.6. Rename Command

When the remote FTP User process sends the command 'Rename' followed by two property lists describing the old and new files, FtpServProt parses the property lists and calls

(CtxRunning>>FtpCtx.Rename)(oldPList,newPList)

which should decide whether to accept the command. The FTP protocol does not require that user access information be present in newPList, so access checking should be done on oldPlist only. To refuse the rename request, Rename should call FTPM(markNo, code, string) and return false. Otherwise it should rename the file returning true if successful. If the rename operation fails, Rename should call FTPM(markNo, code, string) and return false.

File FtpServProtMail.br implements the server part of the Mail Transfer Protocol. This description ignores various critical sections and other vital considerations which must be handled by the user-supplied routines in order to provide a reliable mail service. For the semantics of the protocol see <Pup>MailTransfer.ears.

2.7. StoreMail Command

When the remote FTP User process sends the command 'StoreMail' followed by a property list, FtpServProt parses the property list and calls

(CtxRunning>>FtpCtx.StoreMail)(pList)

which should decide whether to accept the command. To accept, StoreMail need only return true; no property list is sent back in this command. To refuse the command StoreMail should call FTPM(markNo, code, string) and return false, in which case the next procedure (StoreMailFile) is not called.

If StoreMail returns true, FtpServProt tells the User process to go ahead and send the mail, and then calls

(CtxRunning>>FtpCtx.StoreMailFile)(pList) to transfer the file data. When StoreMailFile has finished the transfer, it should return true if everything went OK.

Finally, FtpServProt calls

(CtxRunning>>FtpCtx.StoreMailCleanup)(pList,ok)

where 'ok' is true if StoreMailFile returned true and the User indicated that everything went ok. If 'ok' is false, StoreMailCleanup should delete the file, since it is almost certainly damaged. Note that if StoreMail returned true, StoreMailCleanup will always be called, but StoreMailFile may not. If StoreMail allocates any resources (such as opening a file) they should be deallocated here.

2.8. RetrieveMail Command

When the remote FTP User process sends the command RetrieveMail followed by a property list, FtpServProt parses the property list and calls

(CtxRunning>>FtpCtx.RetrieveMail)(pList)

which should decide whether to accept the request. To refuse, RetrieveMail should call FTPM(markNo, code, string) and return false. To accept, it should return true; no property list is sent back in this command.

If RetrieveMail returns true, FtpServProt then calls

(CtxRunning>>FtpCtx.RetrieveMailFile)(pList)

which should transfer the file. When RetrieveMailFile has finished, it should return true if everything went OK.

Next, FtpServProt calls

(CtxRunning>>FtpCtx.FlushMailBox)(pList)

which should flush the contents of the mailbox. If this operation fails, FlushMailBox should call FTPM(markNo, code, string) and return false, otherwise it should return true.

2.9. MoveMailToFile Commmand

When the remote FTP User process sends the command MoveMailToFile followed by a property list, FtpServProt parses the property list and calls

(CtxRunning>>FtpCtx.MoveMailToFile)(pList)

which should decide whether to accept the request. To refuse, MoveMailToFile should call FTPM(markNo, code, string) and return false. To accept the request, it should perform the operation and return true. If the operation fails, MoveMailToFile should call FTPM(markNo, code, string) and return false.

3. User

The FTP User module (files FtpUserProt.br, FtpUserProtFile.br, and FtpUserProtMail.br) implements the User protocol exchanges.

Many of the procedures in this module report results by returning a word containing an FTP mark code in the right byte and a subcode in the left byte (referred to below as 'subcode,,mark'). Marks and subcodes are the first two arguments to the FTPM procedure which is described in more detail in the Utility section. If the mark type is 'markNo', the subcode describes the reason why the Server refused; your modules may be able to fix the problem and retry the command. The package will output to dspStream text accompanying No, Version, and Comment marks.

3.1. Common User Protocol

File FtpUserProt.bcpl contains routines shared by FtpUserProtFile.br and FtpUserProtMail.br. It uses the bspStream, bspSoc, and dspStream fields in its FtpCtx and contains the following external procedures:

UserOpen(Version) = true|false
    UserOpen should be called after the BSP Connection is open. It sends a version command and aborts the connection returning false if the Server's protocol is incompatible. Otherwise it calls

```
Version()
```

which should generate some herald text:

```
Wss(CtxRunning>>FtpCtx.bspStream, "Alto Pup FTP User ")
```

to which UserOpen will append a string of the form "1.13 15-May-77", and then return true. The herald string received from the Server is output to dspStream.

UserClose(abortIt [false])
    UserClose closes the FTP connection, aborting it if 'abortIt' is true.

UserFlushEOC() = true|false
    flushes bspStream up to the next command, and returns true if it is EndOfCommand. If the stream closes or times out, it returns false. It calls UserProtocolError if it encounters anything except an EOC.

UserGetYesNo(flushEOC) = subcode,,mark
    flushes bspStream up to the next command, which must be 'Yes' or 'No'. If flushEOC is true, it then calls UserFlushEOC and returns the Yes or No mark and accompanying subCode. If the stream closes or times out, it returns false. UserGetYesNo calls UserProtocolError if it encounters anything except Yes or No followed by EOC.

UserProtocolError()
    Writes an error message to dspStream and then calls UserClose to abort the connection.

3.2. User File Operations

File FtpUserProtFile.br implements the User protocol for standard file operations. It uses the bspStream, bspSoc, and dspStream fields in its FtpCtx and contains the following external procedures:

UserStore(pList, StoreFile) = subcode,,mark
    Attempts to send the file described by 'pList' to the remote Server, calling the user-supplied procedure 'StoreFile' to transfer the data. It returns zero if something catastrophic happens (such as the Server aborts the connection), in which case retrying is probably futile.

UserStore sends pList to the Server for approval. The Server can refuse the command at this point, in which case UserStore returns subcode,,markNo. If the Server accepts the command, UserStore calls

StoreFile(pList)

which should transfer the file data. This package provides procedures for transferring data from a disk stream to a network stream, but you are free to write your own. StoreFile should return true if the transfer went successfully. If some environment-specific thing goes wrong (such as an unrecoverable disk error), StoreFile should call FTPM(markNo, code, string, true) before returning false. UserStore then asks the Server if the transfer went successfully and returns subcode,,mark. If mark is 'markYes', the file arrived at the Server safely.

UserRetrieve(pList, Retrieve) = subcode,,mark
Attempts to retrieve the file described by 'pList' from the remote Server, calling the user-supplied procedure 'RetrieveFile' to transfer the data. UserRetrieve returns zero if some catastrophic error occurs, markNo if the Server refuses the command, and markEndOfCommand if the everything goes OK.

UserRetrieve sends pList to the Server and waits for approval. The Server can refuse the command at this point, in which case UserRetieve returns subcode,,markNo. If the Server can handle property lists that specify multiple files, then the following steps are taken for each file:

If the Server has no more files matching the original pList, UserRetrieve returns subcode,,markEndOfCommand (subcode is undefined in this case). Otherwise the Server sends a fully-specified property list describing a file which it is willing to send. UserRetrieve parses this into pList and calls

Retrieve(pList)

which should decide whether to accept the file. To skip the file, Retrieve should return false. UserRetrieve so informs the Server and then loops. To accept the file, Retrieve should return a procedure which UserRetrieve can call to transfer the data. Don't open the file yet, because the Server can still back out, in which case UserRetrieve skips the next step and just loops. If Retrieve returns true, UserRetrieve tells the Server to send the file and then calls

RetrieveFile(pList)

which should open the file, transfer the data, and close the file. This package contains procedures for transferring data from a network stream to a disk stream, but you are free to write your own. When RetrieveFile is done, it should return true if everything went OK. UserRetrieve then loops.

UserDelete(pList,Delete) = subcode,,mark
Requests the remote Server to delete the files described by 'pList', calling the user-supplied procedure DeleteFile before allowing the server to actually delete anything. UserDelete returns zero if some catastrophic error occurs, markNo if the Server refuses the command, and markEndOfCommand if the everything goes OK.

UserDelete sends pList to the Server and waits for approval. The Server can refuse the command at this point, in which case UserDelete returns subcode,,markNo. If the Server can handle property lists that specify multiple files, then the following steps are taken for each file:

If the Server has no more files matching the original pList, UserDelete returns subcode,,markEndOfCommand. Otherwise the Server sends a fully-specified property list describing a file which it is willing to delete. UserDelete parses this into pList and calls

Delete(pList)

which should return true to confirm deleting the file described by 'pList'. UserDelete passes this answer on to the Server and then loops.

UserDirectory(pList, Directory) = subcode,,mark
Requests the remote Server to describe in as much detail as it can files matching 'pList', and then calls the user-supplied procedure Directory when the answers come back.

UserDirectory sends pList to the Server and waits for an answer. The Server can refuse the command at this point, in which case UserDirectory returns subcode,,markNo. If the Server can handle property lists that specify multiple files, then the following steps are taken for each file:

If the Server has no more files matching the original pList, UserDirectory returns subcode,,markEndOfCommand. Otherwise the Server sends a property list which UserDirectory parses into pList and calls

Directory(pList)

and then loops.

3.3. User Mail Operations

File FtpUserProtMail.br implements the user part of the Mail Transfer Protocol. This description ignores various critical sections and other vital considerations which must be handled by the user-supplied routines in order to provide a reliable mail service. For the semantics of the protocol see <Pup>MailTransfer.ears.

UserStoreMail(pList,StoreMail)
Attempts to send mail to the mailbox described by 'pList' at the remote Server, calling the user-supplied procedure 'StoreMail' to transfer the data. It returns zero if something catastrophic happens (such as the Server aborts the connection), in which case retrying is probably futile.

UserStoreMail sends pList to the Server for approval. The Server can refuse the command at this point, in which case UserStoreMail returns subcode,,markNo. If the Server accepts the command, UserStoreMail calls

StoreMail(pList)

which should transfer the mail. StoreMail should return true if the transfer went successfully. If some environment-specific thing goes wrong (such as an unrecoverable disk error), StoreMail should call FTPM(markNo, code, string, true) before returning false. UserStoreMail then asks the Server if the transfer went successfully and returns subcode,,mark. If mark is 'markYes', the mail arrived at the Server safely.

UserRetrieveMail(pList,RetrieveMail) = subCode,,mark
Attempts to retrieve the contents of the mailbox described by 'pList' from the remote Server, calling the user-supplied procedure 'RetrieveMail' to transfer the data. UserRetrieveMail returns zero if some catastrophic error occurs, markNo if

the Server refuses the command, and markEndOfCommand if the everything goes OK.

UserRetrieveMail sends pList to the Server and waits for approval. The Server can refuse the command at this point, in which case UserRetieveMail returns subcode,,markNo. Otherwise UserRetrieveMail calls

RetrieveMail(pList)

which should transfer the file data. When RetrieveMail is done, it should return true if everything went OK.

UserMoveMailToFile(pList) = subCode,,mark
requests the server to move the contents of the mailbox described by 'pList' to the file also described by pList. UserMoveMailToFile returns zero if some catastrophic error occurs, markNo if the Server refuses the command and markYes if everything goes OK.

4. Utility Routines

The utility module (files FtpUtilB.br, FtpUtilA.br, FtpUtilXfer, FtpUtilDmpLd, and FtpUtilInit.br) contains protocol routines shared by the User and Server modules, and some routines for efficiently manipulating disk streams.

InitFtpUtil()
builds some internal tables and streams, getting space from sysZone. You must call this procedure before starting a Server or issuing any User commands.

FTPM(mark, subCode [0], string [], eoc [false], par0, par1, par2, par3, par4)
sends the FTP command 'mark' to the remote FTP process, including 'subCode' if the command requires one, and 'string' if one is present. Then, if 'eoc' is true, an EOC command is sent. 'String' is written to bspStream using the Template package, and may contain imbedded format information. 'Par0' through 'par4' are passed as arguments to the PutTemplate call. The subcode and string arguments further explain certain commands. For markNo, subCode is a machine-readable explanation of why a request was refused, and 'String' is human-readable text such as "UserName and Password required". Codes are tabulated in an appendix to <Pup>FtpSpec.ears. New codes may be registered on request.

GetCommand(timeout [30000]) = subCode,,mark
flushes bspStream up to the next command and returns the mark and subcode (if any). Returns false if the stream closes or it hangs for 'timeout' miliseconds while waiting for a byte. Comment commands are ignored. GetCommand writes the strings accompanying Version, No, and Comment commands to dspStream.

FileType() = Text|Binary
Resets diskStream, scans it (using ReadBlock) looking for high order bits on, and then Resets it again. As soon as it encounters a byte with the high order bit on, it returns 'Binary', otherwise (having read the entire file) it returns 'Text'. This routine uses the DiskStream, buffer, and bufferLength fields in FtpCtx.

The utility module makes three 'process-relative streams' for use by the rest of the package. The only operation defined is 'Puts'.

lst	writes to dspStream
dls	writes to dspStream if debugFlag is true
dbls	writes to bspStream and if debugFlag to dspStream

For example, Wss(dls,string) writes 'string' to the running process' dspStream if the process' debugFlag is set.

4.1. Unformatted Data Transfer

The external procedures below perform efficient operations on disk Streams and use the following fields in FtpCtx: bspSoc, bspStream, dspStream, diskStream, buffer, and bufferLength. The following Alto operating system disk stream procedures are used: SetFilePos, FilePos, FileLength, ReadBlock, WriteBlock, plus the generic stream operations: Gets, Puts, Resets, and Endofs.

DiskToNet() = true|false
Transfers bytes from diskStream to bspStream up to end-of-file, and returns true if everything went OK. Before starting the transfer, DiskToNet outputs "...transferring..." to dspStream, and before returning it outputs "xxx bytes...".

NetToDisk() = true|false
Transfers bytes from bspStream to diskStream until it encounters another FTP command returning true if everything went smoothly. Before starting the transfer, NetToDisk outputs "...transferring..." to dspStream, and before returning it outputs "xxx bytes...".

4.2. Dump Format Data Transfer

File FtpUserDmpLd.br contains two procedures for transferring data between a disk and an FTP connection in dump format. They may be used as the inner loops of the user-supplied data transfer procedures passed to UserStore and UserRetrieve and will create and unbundle dump-format files on the fly. If you don't want to handle dump format, you don't need this file. Dump-file format is described in an appendix to the Alto Executive documentation.

These procedures use the same fields in FtpCtx and the same Alto OS routines as the unformatted transfer routines. Buffer must be at least 130 words long. Making it longer does not speed up the transfer.

DumpToNet(filename []) = true|false
Dumps 'filename' from diskStream to bspStream converting it to dump format, returning true if things go OK. DumpToNet outputs "...xxx bytes" to dspStream before returning. To terminate a dump file, call DumpToNet without a filename.

LoadFromNet() = string or zero
Loads files from bspStream to diskStream (if it is non-zero), converting them from dump format, returning a string when it encounters a name block and zero when it encounters an 'end block'. The caller should not modify the returned string. LoadFromNet outputs "...skipped" or "...xxx bytes" to dspStream for each component file in the dump file.

5. Property Lists

The property list module (files FtpPListProt.br, FtpPList1.br, and FtpPListInit.br) translates between this package's internal representation of a property list and the protocol-specified network representation.

The FTP protocol specifies the syntax of a property list and the syntax of a set of properties sufficient for standard file operations, but states that property lists are extensible. Therefore the property list module comes in two parts: a part that knows the syntax of property lists, and a part which knows the syntax of individual properties. To add new properties you need only modify the latter.

The principal data structure in this module is the File Property List Keyword Table, or fplKT. This table, built by InitFtpPlist, contains (propertyName,propertyObjects) pairs. PropertyNames are strings such as "Byte-size". PropertyObjects know how to Scan (parse) properties into pLists, Generate properties from pLists, initialize properties from a pList full of default values, and Free properties stored in pLists.

5.1. Property List Protocol

File FtpPlistProt.br implements four operations on property lists. This is the module that knows the syntax of a property list, but not the syntax of individual properties. Procedures in this file use the bspStream, bspSoc, and dspStream fields of the FtpCtx and contain the following external procedures:

InitPList(defaultPList []) = pList
    Creates an empty pList, and initializes it to be a copy of 'defaultPList' if one was supplied.

FreePList(pList)
    Destroys 'pList' and returns 0 to facilite writing pList = FreePList(pList). If pList is zero, FreePList returns zero without doing anything.

ScanPList() = pList|false
    Expects to find a property list in bspStream. ScanPList parses this property list and returns a pList if it had proper syntax. If the property list is malformed, ScanPList calls FTPM(markNo, code, string) and returns false. If the connection closes or ScanPList waits for more than 30 seconds while trying to read from bspStream, it returns false.

GenPList(pList)
    Generates a property list in network format from 'pList' and sends it to bspStream.

5.2. The 'Standard' Properties

Files FtpPlist1.br and FtpPlistInit.br implement the standard properties. These files know the syntax of individual properties; they contain the operation procedures for the standard property objects. These files are used by the FTP subsystem and IFS and are sufficient for performing 'standard' file operations. If you wish to add properties, these are the modules which you must change. In addition to the property operations which are rather specialized to their task, there are a few generally useful procedures which are made external:

InitFtpPList()
    which makes the standard property objects and builds fplKT, getting space from

sysZone. This procedure must be called before calling any of the procedures in FtpPlist.br (which typically means before starting a server or calling any procedures in the User module).

Nin(string,lvDest) = true|false
Interprets 'string' as a decimal number and leaves the result in 'lvDest', ignoring leading blanks and terminating on end of string. A null string results in lvDest getting 0. Returns false if the string contains any characters other than 0-9 and <space>.

ParseDate(string,lvRes) = true|false
Parses the string format date into an Alto format date which it puts into the two word vector at 'lvRes'. Returns true if it could parse the date. ParseDate expects the format of the string to bear some similarity to "day-month-year hour:minute:second".

WriteDT(stream,dt)
converts 'dt' from 32 bit Alto date format to a string of the form "dd-mmm-yy hh:mm:ss" and writes it to 'stream'.

6. Revision History

March 30, 1977

First release.

May 15, 1977

Added Directory and Rename commands. Server now handles property lists which specify multiple files. Added User and Server mail operations.

June 8, 1977

Overlay machinery was changed and some bugs were fixed. Some structure definitions changed, so recompilation of user programs is necessary.

July 17, 1977 DiskToNet and NetToDisk moved out of FtpUtilb into a new file FtpUtilXfer. Property lists reorganized, causing changes to the calling interface in FTPSFI. Plist module now uses the Keyword routines in the CmdScan package. Recompilation of user programs is necessary. FtpUserDmpLd renamed FtpUtilDmpLd. Timeouts cleaned up.

Get and set bit fields

This package makes it easy to extract and replace strings of up to 16 bits in a vector of bits. It has no virtues except convenience -- it is neither fast nor compact.

GetBits(Base, BitDisp, Count) -> Value
extracts Count bits starting at bit number BitDisp of the bit vector beginning at word address Base and returns them right-justified as Value. Bit numbering begins with the high-order bit of the first word and continues through the low-order, and then continues in the second word, etc. Here are two examples: GetBits(x, 16, 8) is equivalent to x!1 rshift 8; GetBits(x, 13, 1) is equivalent to (x!0 rshift 2) & 1.

SetBits(Base, BitDisp, Count, Value)
replaces Count bits starting at bit number BitDisp relative to Base with the low-order Count bits of the value Value. (Extraneous high-order bits in Value will be ignored.)

GetBits and SetBits perform no error checks -- if BitDisp is negative, or Count is negative or greater than 16, they will do the wrong thing. Count=0 and Count=16 are OK.

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 inacessible 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).

Bcpl Interrupt Interface

A tiny software package is available that permits Bcpl procedures to be called as a result of hardware interrupts on the Alto. The relevant files are contained in Interrupt.Dm. There are two files, Interrupt.br, which contains code that must always be resident (75 instructions), and InterruptInit.br, which contains code that is required only during initialization of interrupt channels (namely FindInterruptMask and InitializeInterrupt) and may be thrown away after initialization is complete (200 instructions). The sources are contained in InterruptSource.dm, which also includes various command files and InterruptEx.bcpl, the example program given at the end of this writeup. A Nova version of this package is available.

The specification of an interrupt channel is uniformly accomplished with a "mask" that has a one bit for the (Alto) interrupt channel to use. Thus mask=1 is the highest priority channel, mask=#40000 the lowest. (The Alto itself assigns no priorities to channels, but conventions followed both in this package and in the operating system define the priorities as given here.)

mask=FindInterruptMask(trialMask)
This function returns a mask for an ununsed interrupt channel of equal or lower priority than trialMask. It is wise always to use this function to assign interrupt channels, as your channel assignments are then relatively decoupled from ones in software packages you use or in the operating system.

mask=InitializeInterrupt(region, length, mask, proc)
This function initializes and arms the interrupt channel specified by "mask." The "region" parameter points to a block of storage that will be used as stack space for the procedure that is called whenever an interrupt goes off; "length" is the number of usable words in that block of storage. Finally, "proc" is the address of the procedure to call on each interrupt.

The "region" is set up in the following way: the first 15 words hold code and context for saving and restoring state when interrupts occur, the last 4 words are a minimal stack frame from which "proc" is called, and the remaining words in between (a block of size "length"-19) are available for stack frames needed by "proc" and any procedures called by "proc".

The result of the call to InitializeInterrupt is the value of the "mask" argument so as to facilitate use of an actual parameter such as "FindInterruptMask(trialMask)", where "trialMask" is the mask of all channels whose priority is to be higher than the one being initialized.

DestroyInterrupt(mask).
Turns off any interrupt channels represented by one bits in "mask." The interrupt package keeps track of all interrupt channels that the user program has enabled, and sets UserFinishProc in the operating system to execute DestroyInterrupt(userInterruptsEnabled) whenever a finish or abort is done. This cleans up the interrupt system before returning to the operating system (note that the previous value of UserFinishProc is properly saved and restored by this package).

CauseInterrupt(mask)
Initiates an interrupt request on any interrupt channels with one bits on in "mask".

DisableInterrupts(); EnableInterrupts()

These procedures disable and enable the interrupt system. The Alto operating system provides procedures of the same name for the same purpose; the copies in the file Interrupt.Asm are provided in case you Junta the operating system. Note that it is legal for interrupt routines to include calls to DisableInterrupts and EnableInterrupts (or to call procedures that do so), since the interrupt system is turned back on (with lower-priority channels masked out) before the user's Bcpl interrupt procedure is executed.

Example:

The following somewhat senseless example illustrates the use of the interrupt package. It enables two interrupt channels; the high priority one is activitated 60 times a second by vertical interval interrupts; the low priority one is activated every second by the high priority one.

```
external [ Ws; InitializeInterrupt; FindInterruptMask; CauseInterrupt ]
static [ lowChannel; tickCount ]
manifest verticalInterval=#421

let main() be
        [
        let stack1=vec 40
        let stack2=vec 200

// Initialize two interrupt channels
        let high=InitializeInterrupt(stack1, 40,
                        FindInterruptMask(1), HighProc)
        lowChannel=InitializeInterrupt(stack2, 200,
                        FindInterruptMask(high), LowProc)
        tickCount=0
// Arrange vertical interval to cause interrupts on channel "high"
        @verticalInterval=@verticalInterval % high
        while true do loop
        ]

and HighProc() be
        [
        if tickCount eq 0 then
                [
                tickCount=-60
                CauseInterrupt(lowChannel)
                ]
        tickCount=tickCount+1
        ]

and LowProc() be
        [
        Ws("Tick ")
        ]
```

ISF - pseudo random file access package

A package is now available which provides direct access to any page of an Alto disk file by maintaining a run-coded table in core of the disk addresses of the pages. Any number of files, stored on any of the disks which the Alto can accommodate, may be accessed simultaneously. This package was designed for use with the virtual memory (VMEM) package, but is useful in its own right. The ISF package does not call any other packages other than the Alto Operating System.

1. Initialization

InitFmap(MAP, LMAP, FP[, CHECKFLAG, INCREMENT, ZONE, DSK])

Initializes the page table for a file. MAP must point to a block of storage of length LMAP. FP is the file pointer (see the O.S. manual) for the file. InitFmap returns false if LMAP is not large enough to accommodate the page table structure, otherwise true.

If the optional CHECKFLAG argument is supplied and is true, then InitFmap will read the page table from page 1 of the file (if it exists) and check it for validity; also, each time IndexedPageIO extends the page table in core, it will write it back on page 1 of the file. This considerably speeds up subsequent uses of the file through ISF. If CHECKFLAG is omitted or false, no special meaning is attached to page 1 of the file.

If the INCREMENT argument is supplied, it determines the number of pages IndexedPageIO will "read ahead" in the file to augment the page table when this becomes necessary. INCREMENT defaults to 10.

InitFmap and IndexedPageIO require a working buffer capable of holding one disk page; the optional ZONE argument to InitFmap specifies how they will acquire the space for this buffer. ZONE=-1 (the default) causes them to allocate the buffer on the Bcpl stack. Otherwise, ZONE must be a standard allocation zone as described in the Alto O.S. manual. ZONE=0 is equivalent to ZONE=sysZone.

The optional DSK argument points to the DSK structure on which the file resides (see the "Disks and Bfs" section of the O.S. manual for details). DSK defaults to sysDisk, the disk on which files are normally stored.

2. Data transfer

IndexedPageIO(MAP, FIRSTREC, CORE, NUMRECS, WFLAG[, LASTNC])

Transfers NUMRECS pages between the file and core, starting at page FIRSTREC in the file and core address CORE, using MAP to obtain the disk addresses, and extending MAP by scanning the file when necessary. WFLAG=0 means read into core, calling Swat if the requested pages do not exist; WFLAG=-1 means write onto the file, extending the file if necessary; WFLAG=1 means read into core, extending the file if necessary. If LASTNC is supplied with WFLAG=-1 (write), LASTNC will be written into the numChars field of the last page transferred, and if it is less than 2 * the page size, the file will be truncated. IndexedPageIO returns the numChars field of the last page transferred.

Note that the page size is determined by the DSK structure supplied to InitFmap. This means, for example, that NUMRECS=1 will transfer 400b words on a

Diablo Model 31 or 44 disk (the usual Alto disk), but 2000b words on a Trident disk.

WriteFmap(MAP)

Writes the page table on page 1 of the file. As mentioned above, this happens automatically if the CHECKFLAG argument to InitFmap was true.

3. Packaging

The ISF package consists of two binary files: ISFINIT.BR which contains InitFmap, and ISF.BR which contains the other two procedures. ISFINIT.BR may be discarded after use.

KBD - a simple keyboard driver

For programs which do not wish to use the keyboard driver provided by the Alto Operating System, a package is now available which provides a basic keyboard input stream capability. In addition to a character stream for keyboard characters, this package also optionally places mouse button and keyset transitions in the stream, and also provides for calling a user-supplied function at interrupt time when any of a user-selected set of characters appears in the input stream.

The KBD package is written entirely in Bcpl and uses only a few basic facilities of the O.S. (such as MoveBlock) and the Interrupt package.

1. Initialization

KBDinit(Zone [sysZone], extraSpace [0]) -> keystream

Initializes the keyboard handler. The necessary working space (about 150 words, plus extraSpace if any) will be allocated from Zone. KBDinit uses the Interrupt package to allocate an interrupt level for sampling the keyboard, buttons, and keyset on every vertical field interrupt. The extraSpace argument specifies how much extra stack space to allocate for use by the interrupt routine beyond the amount actually needed by routines in the package: this extra space is only needed for trap or overflow procedures (see below). KBDinit returns the new keyboard stream, so a typical use might be

```
        keys = KBDinit(Zone)
```

The package assumes the static location OsBuffer points to a ring buffer structure as described in the O.S. manual.

2. Stream operations

Gets(keystream) -> char

Returns the next character from the stream, waiting until a character is present if necessary.

Endofs(keystream) -> empty

Returns true if there are no characters in the stream's buffer.

Resets(keystream)

Clears the stream's buffer.

Puts(keystream, char) -> notFull

If the stream's buffer is not full, adds char at the end of the buffer just as if it had been typed, and returns true. If the buffer is full, does not add char, and returns false.

3. Other facilities

The KBD package provides a number of other facilities through statics defined in the package. Note that even the procedures mentioned below are defined in this way: for example, if you want to supply a trap procedure, you must do something like

```
        external [ kbdTrapProc ]
        kbdTrapProc = MyKbdTrapProc
```

kbdButtonsOn

This static is initially false. If set to true, mouse button and keyset transitions will be placed in the input stream (unless trapped: see below) just like typed characters. The encoding of these events is as follows:

200b	bottom (right) mouse button DOWN
201b	middle mouse button DOWN
202b	top (left) mouse button DOWN
203b	rightmost keyset key DOWN
...	
207b	leftmost keyset key DOWN
210b	bottom (right) mouse button UP
...	
217b	leftmost keyset key UP

kbdTrapTable
kbdTrapProc(char) -> keepIt

The static kbdTrapTable points to a table of 16 words (allocated from Zone by KBDinit) which is interpreted as a table of 256 bits, one for each possible 8-bit character. When the interrupt routine sees a character whose bit in kbdTrapTable is set, instead of placing the character in the buffer it calls kbdTrapProc(char). If kbdTrapProc returns true, the character is placed in the buffer as usual; if kbdTrapProc returns false, the interrupt procedure assumes that kbdTrapProc has done all the necessary processing. This facility is intended for programs which want to detect interrupt characters even if characters are queued ahead of them in the input buffer. kbdTrapProc is initialized to TruePredicate, which causes all characters to be placed in the buffer regardless of the setting of kbdTrapTable.

kbdOverflowProc(char)

If the interrupt routine finds the ring buffer full, it calls kbdOverflowProc(char). kbdOverflowProc is initialized to Noop, which simply discards the character.

4. Packaging

The KBD package consists of two files, KBDINIT.BR and KBD.BR. KBDINIT.BR contains only the KBDinit procedure, and may be discarded after calling KBDinit. KBD.BR contains all the other facilities described in this memo.

LoadRam

The LoadRam procedure loads a 'packed Ram image' from main memory into the Ram, and optionally performs a 'silent boot' to force one or more tasks into the Ram. LoadRam is derived from the LoadPackedRAM procedure described under 'Packed Ram Images' in the Alto Subsystems manual, and it uses packed Ram images produced by the PackMu program also described therein.

1. Initialization

LoadRam is called in the following manner:

```
     res = LoadRam(RamImage, boot [false])
```

This procedure loads the Ram (if one exists) with a packed Ram image pointed to by RamImage. If the boot argument is true (default = false), the Alto is booted as well. LoadRam returns res<0 if there is no Ram or if booting is impossible because there is no Ethernet interface. Res>0 means that the constant memory in the Alto differs from the constants mentioned in RamImage (the value of res is the number of disagreements). Res=0 indicates that all is well. Once LoadRam has been called, the space occupied by LoadRam and the packed Ram image may be reclaimed.

The format of the RamImage vector is as follows:

```
     RamImage!0: Boot locus vector
     RamImage!1 to !#377: Constants in locations 1 to #377
     RamImage!#400 to !#2377: Instructions in locations 0 to #1777
```

A Ram image in this form is constructed by the PackMu program, which converts a .MB-format file (produced by Mu) into a .BR file that may be loaded with your program. The word described in the PackMu documentation as being used for a version number is actually used to set the boot locus vector (if the boot argument is true).

For example, the Trident controller microcode (TriConMc.Mu) is converted into a Ram image (TriConMc.Br) in the following manner:

```
     Mu TriConMc.Mu
     PackMu TriConMc.Mb TriConMc.Br 77766 DiskRamImage
```

The boot locus vector 77766 specifies that tasks 0, 3, and 17 (Emulator and two Trident disk tasks) be started in the Ram and the rest in the Rom. The optional parameter DiskRamImage specifies that the static pointing to the packed Ram image be named DiskRamImage rather than the default RamImage.

The 'silent boot' is achieved by arranging that the starting location of the emulator task in the Ram (location 0) contain the first instruction of the following sequence:

```
     LOC0:   SWMODE;
             :START;
```

where START is defined to be location 20 (the beginning of the Nova emulator's main loop). These instructions must be contained in the packed Ram image. Then, when the machine is software-booted by LoadRam, the emulator task is started in the Ram (because of the setting of the boot locus vector). The two instructions above merely return control to the main Nova emulation loop in the Rom, thereby bypassing the usual disk boot load sequence.

2. Cleanup

When exiting a program that has micro-tasks active in the Ram, it is considered polite to perform a 'silent boot' to force all tasks back into the Rom. If this is not done, subsequent use of the Ram by another program may cause some running task to run awry.

To do this, simply set the boot locus vector to start only the emulator task in the Ram; then use StartIO to boot the machine. This is accomplished by the statements:

```
SetBLV(#177776)
StartIO(#100000)
```

SetBLV is defined in the LoadRam module, and StartIO in the Operating System.

If you throw away LoadRam at initialization time, performing this cleanup presents a slight problem. One way to solve it is simply to issue the SetBLV call immediately after the LoadRam. The boot locus vector will remain set to this value until the StartIO is issued at cleanup time. The disadvantage of this method is that if the user attempts to boot the Alto manually during execution of the program, the first depression of the boot button will have no effect (a potential source of confusion).

Alternatively, you may include in the microcode the following instruction, located at a fixed place (e.g., 22):

```
LOC22:  RMR←AC0, :LOC0;
```

This code may be invoked at cleanup time by a JMPRAM instruction, as follows:

```
(table [ #61010; #1401 ])(#177776, #22) //JMPRAM(22) sets BLV←AC0
StartIO(#100000)
```

MDI: Multiple Directory Lookups

This package allows a program 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.

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,Buffer,BufferLength)

S is a directory: it must be a disk stream. LookupEntries resets S and then reads through it. NAMEVEC is a vector of CNT strings, the file names. A zero entry in NAMEVEC is simply skipped. PRVEC is a vector of lDV*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).

The optional arguments "Buffer" and "BufferLength" give a core buffer that can be used to buffer the disk stream more efficiently. If these arguments are absent, LookupEntries will obtain a small buffer from the stack.

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

LookupEntries will always find the "most recent" version of all files given in NAMEVEC. The first word of the preamble is smashed with the version number of the file found (zero still implies the file was not found).

Bcpl overlay package

This package enables Bcpl programmers to split up their programs almost painlessly into a core-resident portion and any number of type B overlays (see the Bcpl documentation for the exact meaning of this term), any number of which may be in core at one time. In general no changes whatever are required to the programs themselves: all that need be changed is the loading process (Bldr command to the Executive). The package uses the Alto OS only at the Bfs level and below.

Since this package is designed mostly for people with sophisticated needs, this documentation is somewhat less tutorial than usual for Alto Bcpl software packages. People intending to use the package should be prepared to consult its author.

In the descriptions below, Bcpl procedure descriptions are set off by ** so they will stand out better from the surrounding text.

(5/18/77)

This release adds "special entries" -- overlaid procedures accessed through an extra level of code so that the procedure static doesn't change (see below for details).

(12/8/76)

The only changes in this release are the addition of a new static (OverlayCoreOffset) and an increase in the amount of space required for the overlay descriptor table (odvec argument to OverlayScan).

1. How to load your program

Suppose your program comes in the following pieces: .BR files res1, res2, ..., resn are the permanently resident part; ov1-1, ..., ov1-m are the first overlay (order of overlays, or pieces within an overlay, is unimportant); ov2-1, etc. are the second overlay, and so on. The Bldr command should look roughly as follows:

```
    >Bldr/B res1 ... resn x1/B 0/P ov1-1 ... ov1-m x2/B 0/P ov2-1 ...
```

The names x1, x2, etc. are purely arbitrary names: the presence of the /B is what informs Bldr that a new overlay is beginning.

2. Initializing the overlay package

Before you attempt to call any procedure in an overlay, you must initialize the overlay package. The normal way to do this is to call

** OverlayScan(fptr, odvec, odvsize[, fa, buf, bufsize, fixvec, fixsize, disk, epvec, epsize])

Arguments beyond the third are optional. The arguments have the following significance:

Fptr is the FP for the .Run file which contains the overlays. The Alto OS passes a CFA to your entry procedure (see sec. 3.11 of the Alto OS manual), and this CFA contains as its FP the FP of this .Run file: this is the normal way to get hold of this FP.

Odvec is a table area for the overlay package. OverlayScan initializes this area, and it must stay around and not move during the execution of the program. The space required is 5 words per overlay, plus 3 words per special entry (i.e. 3*epsize), plus 25 words of fixed overhead.

Odvsize is the amount of space you have supplied for odvec.

Fa, if present, is a FA at which OverlayScan should start scanning the .Run file. Normally this will be the FA from the CFA mentioned above.

Buf, if present, is a buffer which OverlayScan will use for reading in the .Run file. The bigger the buffer, the faster OverlayScan will be able to read through the file.

Bufsize is the amount of space you have supplied for the buffer.

Fixvec, if present, is a table area into which the overlay package will store information about the addresses of statics which refer to procedures in overlays. If you supply a fixvec and save somewhere the contents which OverlayScan writes into it, you will be able to bypass OverlayScan entirely on subsequent runs of the program (provided you know somehow that the .Run file hasn't changed or moved on the disk) and use the OverlayInit procedure instead, which doesn't scan the .Run file. The space required for fixvec is 1 word per overlay, plus 1 word per special entry, plus 1 word for each procedure in each overlay, plus 1 word of overhead.

Fixsize is the amount of space you have supplied for fixvec.

Disk is the DSK structure on which the .Run file is stored (see sec. 2 of the "Disks & Bfs" section of the Alto OS manual). It defaults to sysDisk, the disk on which the OS normally stores files.

Epvec, if present, is a vector of addresses (lv's) of procedure statics. Normally, the static for a non-resident procedure contains a trap value when its overlay is not in core, or the entry address when the overlay is in core. This makes it impossible to copy the contents of the static freely into other statics or data structures. However, if the address of the static appears in epvec, the package creates a tiny piece of intermediate code in odvec and sets the procedure static to point permanently to this piece of code. For such procedures, you can pass the contents of the static around at will after calling OverlayScan (or OverlayInit).

Epsize is the number of entries in epvec.

OverlayScan returns -1 if odvsize was too small, or -2 if you supplied a fixvec argument and fixsize was too small. Otherwise, OverlayScan returns the number of words of fixvec actually used, or an arbitrary positive number if there was no fixvec argument.

If you supplied a fixvec and saved the contents of both odvec and fixvec, then you can use the following initialization call in the future:

** OverlayInit(odvec, fixvec[, disk])

which simply initializes all the non-resident procedure statics to their appropriate values and sets up a few internal variables. In this case disk defaults to the value of the disk parameter you gave to OverlayScan, or to the (current) sysDisk if that was defaulted.

3. Operation of the package

The overlay package makes no assumptions about how you wish to allocate core space for overlays. Consequently, you must supply (and declare external) a procedure with the following name and arguments:

** UserReadOverlay(od) -> base

This procedure is called on an "overlay fault", which occurs whenever you attempt to call a procedure in an overlay that is not in core. Od is an "overlay descriptor" which you may pass to various procedures described just below. Your UserReadOverlay procedure is responsible for deciding what overlays or other information to discard from core if necessary, calling ReleaseOverlay if necessary to notify the package of overlays being discarded, reading in the new overlay using ReadOverlay, and finally returning base, the address at which you have read in the new overlay.

UserReadOverlay should first call the procedure

** LockPendingCode()

which scans the Bcpl stack and determines which overlays are currently in the

process of execution and hence are not eligible for being discarded. Then, in the course of deciding which overlay to discard, UserReadOverlay may call
** ReleaseOverlay(od, false) -> ok
which returns true if it is OK to discard the overlay whose descriptor is od. To notify the package that an overlay is actually being discarded, call
** ReleaseOverlay(od, true)
In order to discover which overlays are present in core, UserReadOverlay may call
** GeneratePresentOverlays(proc)
which calls proc(od) for each overlay currently in core.

UserReadOverlay may use the following procedures to discover various useful parameters of a given overlay:
** OverlayFirstPn(od) -> pn
returns the page number in the .Run file at which a given overlay begins (the first argument to ReadOverlay, below).
** OverlayNpages(od) -> npages
returns the number of pages required for the overlay on the .Run file and in core (the third argument to ReadOverlay).
** OverlayDiskAddr(od) -> da
returns the disk address of the first page of the overlay.
** OverlayCoreAddr(od) -> base
returns the current core address of an overlay, or 0 if the overlay is not currently in core.

When UserReadOverlay has finished making any necessary decisions, it should call
** ReadOverlay(pn, base, npages)
which actually calls the Bfs to read the overlay into core.

The overlay package supplies three other procedures which likely to be of lesser interest:
** GenerateOverlays(proc)
calls proc(od) for every overlay regardless of whether it is in core or not. This may be useful during initialization when deciding how much space to allocate in core for reading in overlays.
** FindOverlayFromPn(pn) -> od
finds an od given the first page number in the .Run file, or calls Swat if pn is not such a page number.
** DeclareOverlayPresent(od, base)
tells the package to believe that the given overlay is present in core at the given address. (The package automatically calls DeclareOverlayPresent(od, UserReadOverlay(od)) in the course of processing an overlay fault.)

The overlay package also supplies a static which is useful if you are using it in conjunction with the VMEM package. This static is called
OverlayCoreOffset
and is the displacement within the overlay descriptor of the word which holds the core address of the overlay (returned by OverlayCoreAddr). This makes it possible to say things like LockCell(od+OverlayCoreOffset).

4. Restrictions and caveats

There are two known restrictions on use of this package. One is that a procedure in an overlay which is called from outside that overlay must not have more than 20 arguments. The other is a little subtler. Because the package operates by placing a trap value in the static cells of procedures in overlays not present in core, and re-executes the procedure call instruction after bringing in the overlay, the following kind of code will not work:

```
        SavedProcAddr = NonResidentProc
        .
        .
        SavedProcAddr(args)
```
because the package has no way of fixing up SavedProcAddr to point to the core address of the procedure. Because of the way the Bcpl compiler chooses to do things, the same is unfortunately true of the following code sequence:
```
        .
        .
        SavedLvProcAddr = lv NonResidentProc
        .
        .
        (@SavedLvProcAddr)(args)
```
If you need to do this kind of thing (e.g. in a command processor which saves addresses of command procedures, some of which may be non-resident, in a data structure), you should use the epvec and epsize arguments to OverlayScan to declare which procedures need to be accessible this way.

You may also run into trouble if you have a non-resident procedure which uses strings or tables: since these are stored in the code itself, non-resident procedures will have to copy such strings or tables into resident storage if they may be used when the procedure is not in core.

5. Multiple contexts

If you have multiple contexts (in the sense of the Bcpl Context package), it is all right for context switching to occur while control is inside the overlay package itself; in particular, since ReadOverlay calls the Bfs, it is all right for this call on the Bfs to call Block while waiting for the disk. However, the overlay package does assume it will not be pre-empted, i.e. it only allows for context switching during calls on the user-supplied procedure UserReadOverlay and during the Bfs call in ReadOverlay.

If you have more than one context which uses overlays, then when an overlay fault occcurs you must call
```
**      LockPendingCode()
```
to lock any overlays on the current stack, and then
```
**      LockPendingCode(topframe)
```
with the topmost stack frame of each context that might use overlays. LockPendingCode assumes that each stack is allocated downward in core: if you have a stack that violates this assumption, you must sequence through the stack yourself and call
```
**      LockPendingPc(pc)
```
with each saved return address.

6. Use of the package with Trident disks

All page numbers (the page number in the fa argument to OverlayScan, the result of OverlayFirstPn, and the pn argument to ReadOverlay and FindOverlayFromPn) and all page counts (the result of OverlayNpages and the npages argument to ReadOverlay) refer to the sector size of the disk on which the overlay file is stored, i.e. 400b words for the Diablo disks but 2000b words for Tridents. This is consistent with the meaning of "page" for the Bfs and Tfs.

Type B overlays are carefully arranged in .Run files so that they start at page boundaries. You cannot simply copy a .Run file to a Trident and have this property be true with respect to the larger sectors size -- you must insert blank pages in the file as necessary. However, since OverlayScan doesn't look at any part of the file before the fa you give it, you don't need to copy the resident part of the .Run file, only the overlay part; then you can tell OverlayScan to start scanning at page 1 (the first data page).

7. Files

The overlay package consists of the following files:
OverlaysInit.BR - the initialization procedures of section 2 above.
Overlays.BR - the procedures of section 3 above.
OverlaysVmem.BR - some routines for interfacing to the software virtual memory package (VMEM), not described here.
You may discard OverlaysInit after calling the initialization procedures. Needless to say, neither of these files may itself be loaded as part of an overlay.

Paper Tape Package

No computer is complete without paper tape equipment. This package provides standard stream interfaces to a DG Nova High Speed Reader and Punch via the Diablo printer interface. The hardware only works on Alto Is, and only with the particular paper tape equipment we have at Parc.

The package consists of a single binary file, PaperTape.br. The source for this, PaperTape.bcpl, is included in PaperTape.dm, which also contains a test program, PaperTapeTest.bcpl, which generates various test patterns for tuning the punch. Since the punch is mechanical, it must be oiled, and have its levers bent now and then or it stops working.

Besides using standard operating system facilities, this package makes use of the Context and Timer packages. If you don't want to include the Context package, define an external procedure Block() that returns immediately.

There is one externally-callable procedure for the punch stream, which works as follows:

CreatePunchStream(zone [sysZone], leaderLength [50.]) = ptps
    Creates a Paper Tape Punch Stream (ptps) using the supplied parameters, both of which are optional. LeaderLength is the length in inches of leader/trailer (blank tape with only sprocket holes punched) that will be generated when the stream is created, closed or reset. The zone argument specifies the zone from which the stream structure will be allocated (about 15 words). CreatePunchStream turns on the punch, waits 2 seconds for the motor to come up to speed and then punches some leader.

The following operations are defined on a Paper Tape Punch Stream:

Puts(ptps, char)
    Punches the specified 8-bit character (ignoring bits 0-7). Puts does some rather critical timing while punching the character, and so it turns off interrupts for about 4.5 ms. If the punch does not supply a sync signal within a reasonable time, Errors(ptps, ecPunchNotReady) is called.

Resets(ptps)
    Punches some leader. The amount is 50 inches (the default), or the amount specified in the optional second argument to CreatePunchStream.

Closes(ptps)
    Punches some leader, waits 1 second, turns off the punch motor, and then destroys the stream. This includes returning the stream structure to the zone from which it was allocated.

There is one externally-callable procedure for the reader stream, which works as follows:

CreateReaderStream(zone [sysZone]) = ptrs
    Creates a Paper Tape Reader Stream (ptrs). The zone argument specifies the zone from which the stream structure will be allocated (about 15 words). CreateReaderStream releases the brake and capstan so that you can load the tape.

The following operations are defined on a Paper Tape Reader Stream:

Gets(ptps, stop [false]) = char or -1

Reads the next 8-bit character from the tape, returning -1 if the tape runs out. Gets does some rather critical timing while reading the character, and so it turns off interrupts for a while. Unless stop is true, the capstan will be left engaged, and you must call gets before the next character arrives or it will be lost. Resetting the stream will also stop the tape.

Resets(ptps)
Stops the tape and then releases the brake.

Closes(ptps)
Stops the tape, releases the brake, and then destroys the stream. This includes returning the stream structure to the zone from which it was allocated.

WARNING: until the paper tape reader stream is created, the reader is in rip-tape mode: capstan and brake are both engaged!

Pup Event Report Server

This package (file PupERPServ.br) implements a Pup Event Report Server -- a process that listens for Event Report packets and writes them to a file. It will run on Altos and Novas, and uses the Pup package through level 1 (plus the packages that the Pup package uses, in particular the Context package). The server runs as a context (in the sense of the Context package), and you can start up as many instances of the server as you wish, each listening on a different socket and writing to a different file. To instantiate a server call

CreateERPServer(zone, ctxQ, port, diskStream)
: which will create a server and queue it on 'ctxQ', getting space from 'zone' (approximately 1000 words). The server will listen on 'port' for event reports and append them to 'diskStream' (that is, it will positon diskStream to the end and then start writing event entries).

Stopping a cloud of these servers is accomplished by two statics which the user must define:

quitCount	which is incremented for each server started
quitFlag	which all servers watch

The idea is to initialize quitFlag to false and quitCount to zero. When finishing, set quitFlag to true and Block until quitCount goes to zero, then finish.

The event file is a sequence of entries with the following format:

entry length	2 bytes - including these two
event Pup source port	6 bytes
event Pup ID	4 bytes
event Pup contents	remaining bytes

Pup Package

The Pup package consists of a large body of Alto software that implements communication by means of Pups (Parc Universal Packets) and Pup-based protocols. This software is broken into a number of independent modules implementing various "levels" of protocol in a hierarchical fashion. Each level depends on primitives defined at lower levels, and defines new primitives (e.g, inter-network addressing, process-to-process connections, byte streams) available to levels above it. A program making use of the Pup package need include only those components implementing primitives utilized by that program.

1. Overview

This document is organized as a general overview followed by descriptions of each of the components of the package, with the lowest levels described first. A history of revisions to the package may be found at the end.

Before beginning the real documentation, we should like to mention a number of points worth bearing in mind throughout, as well as various caveats and suggestions for use.

a. This document concerns itself only with external program interfaces and not with protocol specifications, internal implementation, motivations for design choices, etc. The Pup package implements the protocols described in the memo "Pup Specifications" (Maxc file <Pup>Pup.Ears) and in other documents also to be found in the <Pup> directory. Users interested in protocol information are referred to those documents. Knowledge of these protocols is not required when writing programs making use of the higher-level primitives provided by the Pup package (specifically, connections and byte streams), but is essential when dealing directly with the lower-level primitives.

b. Since both the software and the protocols are still under active development, users are requested to avoid making changes to the package, if at all possible. This is so that subsequent releases of the package may be incorporated into existing programs with minimum fuss. We have attempted to provide as general-purpose a package as is reasonable (consistent with clean programming practices and considering Alto memory limitations), so if you come up with an application that simply can't be accomodated without modifying the package, we would like to know about it. There are a small number of parameters that we have designated as "user-adjustable" and separated out into a special declaration file (PupParams.decl). The intention is that users be able to change these parameters and recompile the package; however, one should be aware that the software has not been tested with parameters set to values other than the ones in the released version.

c. One of the design goals has been to implement software that will also run on a Nova. All Alto-specific code has been carefully separated out into modules containing "Al" in their names (e.g., PupAlEth.bcpl for the Alto Ethernet driver). The Nova equivalents of the Alto-specific modules (released as a separate package) contain "Nv" in their names. Source files not containing "Al" or "Nv" in their names may be recompiled on the Nova (with BCPL or the Nova version of Altoasm) and run without change; either they are completely free of machine dependencies or (in a few cases) they enclose machine-dependent code in conditional compilation. People writing general-purpose subsystems making use of this package are encouraged to adopt the same approach.

d. The Pup package makes extensive use of primitives provided in four other packages, all of which have been released recently. These are the Context, Interrupt, Queue, and Timer packages. The dependence on the Context package means that calling programs must operate in a manner compatible with contexts. In particular, the Pup package initiates a number of independent background processes that must be given an opportunity to run fairly frequently. Hence, the user's "main program" must run within a context, and wait loops and very long computations in the main program should be interspersed with calls to Block. For example, a call such as "Gets(keys)" (which causes busy-waiting inside the operating system) might be replaced by something like "GetKeys()", where the latter function is defined as:

```
let GetKeys() = valof
    [
    while Endofs(keys) do Block()
    resultis Gets(keys)
    ]
```

Consult the the Context Package writeup for further information.

e. The Pup package operates only under the new operating system, versions 2/0 or higher.

1.1. Organization

The Pup software is divided into three major levels, corresponding to levels 0 through 2 of the Pup protocol hierarchy. Software at a given level depends on primitives provided at all levels below it.

At level 0 is the "transport mechanism" software necessary for an Alto to send and receive Pups on an Ethernet. This consists of a small Ethernet interrupt handler that appends received Pups to an input queue and transmits Pups taken from an output queue. It is the only portion of the Pup package specific to the Ethernet or to the Alto-Ethernet interface.

Level 1 defines a number of important and generally useful primitives. A program desiring to send and receive "raw Pups" (without sequencing, retransmissions, flow control, etc.) would interface to the Pup package at this level. The level 1 module includes the following:

a. Procedures for creating, maintaining, and destroying a "socket", a process's logical connection to the Pup inter-network.

b. Procedures for managing "Packet Buffer Items" (PBIs), each of which holds a Pup and some associated information while the Pup resides in Alto memory.

c. A background process that distributes received Pups to the correct sockets. This includes checking port address fields and optionally verifying the Pup checksum.

d. Procedures for allocating PBIs, building Pups, and queueing them for transmission.

e. A background process that dynamically maintains a routing table for transmission of Pups to arbitrary inter-network addresses.

f. Optional procedures permitting the local host to be a gateway (not ordinarly used).

At level 2 are modules implementing three higher-level protocols: the Rendezvous/Termination Protocol (RTP), the Byte Stream Protocol (BSP), and the Name Lookup Protocol. These are independent, parallel protocols, each built on top of the primitives defined at level 1; however, the RTP and the BSP interact in a way such that, in this implementation, BSP depends on the existence of RTP.

The RTP module contains procedures for opening and closing a "connection" with a foreign process. These have options permitting the local process to operate in the role of either "initiator" or "listener".

The BSP module contains mechanisms for sending and receiving data by means of error-free, flow-controlled "byte streams" between a local and a foreign process. These are true "streams" in the sense defined by the Alto operating system. Additionally, means are provided for sending and receiving Marks and Interrupts, which are special in-sequence and out-of-sequence signals defined by the Byte Stream Protocol. A separate, optional module permits sending and receiving blocks of data in memory an order of magnitude more efficiently than by use of the basic "Puts" and "Gets" operations.

The Name Lookup module contains a procedure for parsing an inter-network "name" (e.g., a host name) and converting it to an address. When necessary, it finds and interacts with some name lookup server on the directly connected network.

1.2. File Conventions

The Pup package is distributed as file PupPackage.dm, which contains the following binary files:

Level 0	
PupAlEthb.br	Alto Ethernet driver (BCPL portion)
PupAlEtha.br	Assembly code for Ethernet driver
PupAlEthInit.br	Alto Ethernet initialization
Level 1	
Pup1b.br	Main level 1 code (BCPL portion)
PupAl1a.br	Assembly-language code for level 1
PupRoute.br	Routing table maintenance and access
PupDummyGate.br	Dummy substitute for gateway code
Pup1Init.br	Level 1 initialization
Level 2	
PupRTP.br	Rendezvous/Termination Protocol
PupBSPStreams.br	Byte Stream Protocol (BCPL portion)
PupBSPProt.br	Additional BSP code
PupBSPa.br	Assembly-language code for BSP
PupBSPBlock.br	Fast BSP block transfer procedures
PupNameLookup.br	Name lookup module

The files with "Init" in their names contain initialization code that need be executed only once and may then be thrown away.

Additionally, the following "get" files are included. They contain declarations of all structures and other parameters likely to be of interest to calling programs (as well as some others of no interest to callers). We suggest that the user make listings of these files to accompany this documentation.

Pup0.decl	Level 0 definitions (network-independent)
Pup1.decl	Level 1 definitions
PupRTP.decl	Definitions for RTP
PupBSP.decl	Definitions for BSP
Pup.decl	Does "get" of all the above
PupParams.decl	User-adjustable parameters
PupStats.decl	Statistics definitions
PupAlEth.decl	Definitions specific to Alto Ethernet

A program that does a "get" of any of the first group of files must also "get" all files earlier on the list, and in the same order. (We were not able to make this happen automatically because of a limit on the number of simultaneous open files at compilation time). The file Pup.decl is provided for the convenience of programs dealing with the package at the BSP level. A "get" of PupParams.decl is included in Pup0.decl, and PupAlEth.decl is not ordinarily of interest to outside programs.

The following table shows, for each module (including external packages), what .br files constitute that module and what other modules are also required.

Module Name	Files	Other Modules Required
BSP Block Transfer	PupBSPBlock.br	BSP ByteBlt
ByteBlt (external)	AltoByteBlt.br	
BSP	PupBSPStreams.br PupBSPProt.br PupBSPa.br	RTP
RTP	PupRTP.br	Level 1
Name Lookup	PupNameLookup.br	Level 1
Level 1	Pup1b.br PupAl1a.br PupRoute.br PupDummyGate.br Pup1Init.br	Level 0 Timer
Level 0	PupAlEthb.br PupAlEtha.br PupAlEthInit.br	Context Interrupt Queue
Context (external)	Context.br ContextInit.br	
Interrupt (external)	Interrupt.br InterruptInit.br	
Queue (external)	AltoQueue.br	
Timer (external)	AltoTimer.br	

For debugging purposes, a version of the package compiled with the "pupDebug" conditional enabled is distributed as PupDebug.dm. It includes some additional consistency checking at the cost of space and time.

The sources for the Pup package are contained in file PupSources.dm, and consist of the following files:

PupAlEthb.bcpl	PupAlEtha.asm	PupAlEthInit.bcpl
Pup1b.bcpl	PupAl1a.asm	Pup1Init.bcpl
PupRoute.bcpl	PupDummyGate.bcpl	
PupRTP.bcpl		
PupBSPStreams.bcpl	PupBSPProt.bcpl	PupBSPa.asm
PupBSPBlock.bcpl		
PupNameLookup.bcpl		

Additionally, there are several command files:

CompilePup.cm	Compiles all the source files
CompilePupDebug.cm	Compiles with "pupDebug" enabled
DumpPupPackage.cm	Creates PupPackage.dm
DumpPupDebug.cm	Creates PupDebug.dm
DumpPupSources.cm	Creates PupSources.dm
Pup.cm	A list of all the source files

The source files are formatted for printing in a small fixed-pitch font such as Gacha8 (as used by the command "Gears/s").

1.3. Glossary of Data Types

Name	Defined in	Meaning
BSPSoc	PupBSP.decl	BSP-level Pup socket, consisting of an RTP socket (RTPSoc) followed by additional information about a byte stream. This includes byte IDs (sequence numbers), queues, and allocations for incoming and outgoing data and interrupts, and a BSP stream block (BSPStr).
BSPStr	PupBSP.decl	BSP stream (part of a BSPSoc), for interfacing the BSPSoc to the Alto operating system's stream mechanism.
HTP	Pup1.decl	Hash Table Preamble, defining the publicly-accessible operations on a hash table object (specifically, the Pup routing table).
NDB	Pup0.decl	Network Data Block, containing information specific to each network physically attached to the local host (the Alto has only one of these, namely etherNDB).
PBI	Pup0.decl	Packet Buffer Item, which holds a Pup and various associated information.
PF	Pup0.decl	Packet Filter, controlling acceptance of incoming packets on a given network.
Port	Pup0.decl	An inter-network address, consisting of network, host, and socket numbers, as defined by protocol.
PSIB	Pup1.decl	Pup Socket Info Block, contains data used for setting initial default values when a PupSoc is created.
Pup	Pup0.decl	An inter-network packet, as defined by protocol.

PupSoc	Pup1.decl	Level 1 Pup socket, defining a process's logical connection to the inter-network. It contains default local and foreign port addresses, PBI allocation information, and an input queue header.
RT	--	Routing Table, containing information necessary to route outgoing Pups to destination hosts or to gateways. There is only one instance of an RT, called pupRT. The structure of an RT is not public, but procedures are provided for accessing and enumerating individual Routing Table Entries (RTEs), which are public structures.
RTE	Pup1.decl	Routing Table Entry (routing information for one network).
RTPSoc	PupRTP.decl	RTP-level Pup socket, consisting of a level 1 socket (PupSoc) followed by additional information about a connection. This includes state, connection ID, timers, and a higher-level Pup-handling procedure.
soc	--	An instance of a PupSoc, RTPSoc, or BSPSoc, depending on context. Note that a PupSoc may be the initial portion of an RTPSoc, which may in turn be the initial portion of a BSPSoc; hence, a given soc may be an instance of more than one of these structures.
str	--	An instance of a stream (most likely, a BSPStr).

2. Level 0 Interface

The level 0 module (files PupAlEthb, PupAlEtha, and PupAlEthInit) serves only to interface the Alto Ethernet to the network-independent Pup level 1 module. Assuming the level 1 code is being used, as is normally the case, external programs will generally have no occasion to deal directly with the level 0 module. Provisions are also made for sending and receiving non-Pup Ethernet packets, for use in unusual applications.

This module requires the existence of the following external statics (all of which are defined in level 1):

ndbQ	A pointer to a two-word queue header (hereafter referred to as "a queue"; see Queue Package documentation) upon which the Ethernet NDB (etherNDB) may be queued by this module. In a machine with more than one network interface (e.g., a Nova), this queue would contain an NDB for each network.
pbiFreeQ	A queue from which free PBIs may be obtained, for buffering received Pups.
pbiIQ	A queue to which PBIs are appended when Pups are received.
lenPup	The maximum length of a Pup (in words).

The externally-callable procedures in this module are the following:

InitAltoEther(zone, ctxQ)
Initializes the Alto Ethernet interface and associated data structures. "zone" is a

free-storage zone from which space may be obtained for permanent data structures (currently less than 100 words). "ctxQ" is a queue on which a context created by this procedure may be queued. This procedure allocates an NDB and appends it to ndbQ; allocates an interrupt context (see Interrupt Package documentation) and sets it up to field Ethernet interrupts; and allocates and initiates an ordinary context (see Context Package documentation) which runs forever and whose job it is to restart the Ethernet interface if it is ever shut off due to running out of free PBIs for input. InitAltoEther returns having done nothing if the Alto doesn't have an Ethernet interface installed (the level 1 initialization detects the condition of ndbQ being empty after all interface initialization procedures have been called).

EncapsulateEtherPup(pbi,pdh)
Encapsulates the Pup contained in "pbi" for transmission to physical destination host "pdh" on the directly-connected Ethernet. The PBI should contain a completely well-formed Pup. EncapsulateEtherPup sets the Ethernet destination, source, and type fields in the encapsulation portion of the packet, and also sets the packetLength word in the PBI. SendEtherPup is the procedure called from level 1 via the encapsulatePup entry in the Ethernet NDB.

SendEtherPacket(pbi)
Queues "pbi" for transmission on the directly-connected Ethernet, and initiates transmission if the interface is idle. The PBI should contain a completely well-formed Ethernet packet (which need not be a Pup). the packetLength word in the PBI must contain the physical length of the packet in words, and pbi>>PBI.queue must contain a pointer to a queue to which the PBI will be appended after it has been transmitted. SendEtherPacket is the procedure called from level 1 via the level0Transmit entry in the Ethernet NDB.

SendEtherStats(pbi,ndb) = true or false
If the debugging version of PupAlEth is loaded (pupDebug on), this procedure copies the statistics accumulated by the Ethernet interface (described by ndb) into pbi and returns true. If the module was not compiled with debugging on, SendEtherStats immediately returns false.

When a packet is received from the Ethernet, the input interrupt routine first verifies that the hardware and microcode status are correct, and discards the packet without error indication if not. It then tests the packet for acceptance by each Packet Filter (PF) on the Ethernet packet filter queue, as will be described shortly. If some PF accepts the packet, the PBI is then enqueued on the queue designated in the PF; otherwise it is discarded. A free PBI is then obtained from pbiFreeQ, and the receiver is restarted. (Actually, an attempt is made to restart the receiver before any other processing so as to minimize the interval during which a packet could be missed because the receiver isn't listening to the Ethernet.)

When an output PBI is passed to SendEtherPacket, it is queued on a local Ethernet output queue (eOQ). If the interface is currently idle, transmission is initiated immediately; otherwise, the PBI is simply left on the queue for action by the interrupt routine. When an output completion interrupt occurs (or a fatal error indication such as a "load overflow", or a 100 millisecond software timeout), the PBI is then enqueued on the queue specified in the PBI (typically pbiFreeQ or a level 1 queue called pbiTQ).

Garden-variety errors (e.g., collisions, bad Ethernet CRCs, etc.) are handled automatically: input errors cause the received packet simply to be discarded, while output errors cause retransmission. "Impossible" errors (suggesting that the interface or the Alto is broken) result in a call to SysErr(@ePLoc,ecBadEtherStatus).

In the debugging version of this module (pupDebug on), a number of Ethernet performance statistics are gathered. These are intended for experimental purposes and measurements. One should consult PupAlEthb.bcpl to see what is collected.

Though the primary purpose of the Pup level 0 module is to send and receive Pups on a particular directly-connected network, means are also provided for sending and receiving arbitrary network-dependent packets (i.e., Ethernet packets in an Alto).

Sending a non-Pup packet is straightforward: one simply calls SendEtherPacket after constructing the desired Ethernet packet in the PBI, as described above.

Discrimination among received packets is accomplished by one or more objects called Packet Filters (PFs), which reside on a Packet Filter Queue (pfQ) whose head is in the NDB. Each PF contains a predicate and a pointer to a queue. When a packet is received, the predicate in each PF in turn is called with the PBI as an argument. If the predicate returns true, the PBI is enqueued on the queue pointed to by the PF; if it returns false, the next PF is tried. If no PF accepts the packet, the PBI is discarded.

The pfQ initially contains a single PF that accepts Pups and appends them to pbiIQ (the level 1 Pup input queue). A program desiring to receive other kinds of Ethernet packets should construct its own PF and enqueue it on the Ethernet pfQ.

3. Level 1 Interface

The level 1 module (files Pup1b, PupAl1a, PupRoute, PupDummyGate, and Pup1Init) contains the mechanisms enabling a process to send and receive individual Pups to and from other processes at arbitrary inter-network addresses. Concepts such as "connection" and "stream", however, are not defined at this level, so it is the process's responsibility to perform initial connection, sequencing, retransmission, duplicate detection, etc., where required.

A process deals with the level 1 module through a PupSoc, a level 1 socket structure (see Pup1.decl), which completely describes that process's interface to the inter-network at the first level of protocol. The information in the socket is as follows:

iQ	Input queue. PBIs received for the socket are appended to this queue. The two-word queue header is included in the socket structure itself, so to remove a packet from the iQ one would write "Dequeue(lv soc>>PupSoc.iQ)".
lclPort	Local port address (a Port structure). This serves two purposes. First, the "socket number" in the port enables the level 1 Pup input handler to distribute each incoming Pup to the correct PupSoc by comparing pbi>>PBI.pup.dPort.socket (the Pup destination socket number) with soc>>PupSoc.lclPort.socket of each active PupSoc until a match is found. Second, the source port fields of each outgoing Pup generated by the process are defaulted (if zero) to the values given in the local port address.
frnPort	Foreign port address (a Port structure). This provides information for defaulting the destination port fields of outgoing Pups, in the same manner as described for lclPort.
psib	Pup Socket Info Block (PSIB), which contains the information described below. Since it is generally the same for all sockets, there is a "default PSIB" (dPSIB) whose contents are copied into the psib for each socket when the socket is created.

maxTPBI	The maximum total number of PBIs that may be assigned to the socket. Since free PBIs are taken from a common pool, some means is required for ensuring that no single socket can usurp more than a certain share of the total available PBIs (which, aside from reducing performance for other sockets, could lead to deadlocks in higher-level protocols if the free pool became exhausted). This is discussed further in the descriptions for the GetPBI and ReleasePBI procedures.
numTPBI	The total number of additional PBIs that may be assigned to the socket (i.e., maxTPBI minus the number of PBIs already assigned).
maxIPBI	The maximum number of PBIs that may be assigned to the socket for input use.
numIPBI	The number of additional PBIs that may be assigned for input (i.e., maxIPBI minus the number of PBIs already assigned for input).
maxOPBI	The maximum number of PBIs that may be assigned to the socket for output use.
numOPBI	The number of additional PBIs that may be assigned for output (i.e., maxOPBI minus the number of PBIs already assigned for output).
doChecksum	If true, the Pup software checksum is checked by the level 1 software in incoming Pups (before being given to the process) and generated in outgoing Pups. The default value is true.

The following statics are defined within the level 1 module and may be referenced externally (though only a few are likely to be of interest):

ndbQ	Pointer to queue of NDBs for all the physically connected networks (see level 0 description). The first NDB on ndbQ is considered to be the "default" network, i.e., the one sent to if a process specifies a Pup destination network of zero.
numNets	The number of directly connected networks (always 1 in an Alto).
pbiFreeQ	Pointer to queue of free PBIs.
pbiIQ	Pointer to queue on which incoming Pups are placed by level 0 interrupt routines.
pbiTQ	Pointer to queue on which outgoing Pups are ordinarily placed after transmission.
gatewayIQ	Pointer to queue on which received Pups not addressed to this host are placed. Unless the Gateway package is loaded, gatewayIQ is initialized to pbiFreeQ.
socketQ	Pointer to queue of all active PupSocs.
pupRT	Pointer to routing table (described later).
dPSIB	Pointer to default socket info block, used to provide initial values in part of each PupSoc when it is created.

pupZone	Default zone from which allocations will be made by the Pup package. This is initialized to the "zone" argument to InitPupLevel1.
pupCtxQ	Default context queue onto which new contexts created by the Pup package will be appended. This is initialized to the "ctxQ" argument to InitPupLevel1.
maxPupDataBytes	The maximum number of data (content) bytes in a Pup. This is initialized to the "pupDataBytes" argument to InitPupLevel1 and remains constant thereafter.
lenPup	The length of the largest possible Pup, in words (derived from maxPupDataBytes).
lenPBI	The length of a PBI, in words (derived from lenPup). Note that all PBIs are of the same size and can each contain a Pup of maximum length.

The level 1 module must be initialized by calling InitPupLevel1, as follows:

InitPupLevel1(zone, ctxQ, numPBI, pupDataBytes [defaultPupDataBytes])
Initializes all the level 1 software, and also calls the appropriate level 0 initialization (InitAltoEther in the Alto version). "zone" is a free-storage zone from which permanent allocations may be done. "ctxQ" is a pointer to a queue of contexts to which the contexts created by this procedure may be appended. "numPBI" is the number of PBIs to be allocated (from "zone") and appended to the pbiFreeQ. The optional argument "pupDataBytes" specifies the maximum number of data (content) bytes to be permitted in any Pup; it must be even and by convention should not be greater than 532. A smaller maximum Pup length is useful in some applications not requiring high throughput, since the PBIs are thereby smaller and one can have more of them at the same cost in memory. The default value of this parameter is 532.

InitPupLevel1 does the following: it creates the queues pbiIQ, pbiTQ, pbiFreeQ, socketQ, and ndbQ; allocates "numPBI" PBIs and appends them to pbiFreeQ; creates the routing table pupRT; creates the default Pup socket info block dPSIB; calls the level 0 initialization procedure(s); creates the PupLevel1 and GatewayListener background contexts (to be described later); and broadcasts requests for gateway routing information. The total amount of storage taken from "zone" (in words) is approximately numPBI*lenPBI + lenPSIB + lenPupSoc + 300 + the amount needed by level 0 initialization. InitPupLevel1 also initializes the static pupZone to "zone" and pupCtxQ to "ctxQ", and sets up the constants maxPupDataBytes, lenPup, and lenPBI on the basis of "pupDataBytes".

InitPupLevel1 does not call Block, so it is permissible to call it from initialization code that is not running as a context.

The following procedures are provided for creating and destroying sockets:

OpenLevel1Socket(soc, lclPort [defaulted], frnPort [zeroes])
Creates a PupSoc. "soc" should point to a block of size lenPupSoc. "lclPort", if specified and nonzero, points to a Port structure describing the desired local port address. "frnPort", if specified and nonzero, points to a Port structure describing the desired foreign port address. The "soc" is then appended to socketQ, thereby enabling reception of Pups directed to it.

Each field in the local port address is subject to defaulting if either the "lclPort" is unspecified or the field is zero, in the following manner. If the socket number is unspecified, one is chosen at random (it is guaranteed unique).

If both the network and host numbers are unspecified, they are filled in with a reasonable local host address (perhaps based on the supplied "frnPort"). Ordinarily, one should allow the socket number to be defaulted unless one intends the process to reside at a "well-known socket" (as in a server), and one should always allow the network and host numbers to be defaulted.

If "frnPort" is unspecified, the foreign port in the "soc" is set to zeroes. Then, if the foreign network number is zero (generally for the purpose of designating the "directly connected" network), it is set to the connected network's actual number, if known. Note that the "lclPort" and "frnPort" fields in the "soc" are copied from the corresponding arguments to OpenLevel1Socket; the argument ports are not modified and are not needed thereafter.

CloseLevel1Socket(soc)
Causes "soc" to be removed from socketQ. This procedure blocks until all PBIs assigned to the socket have been recovered and released. If "soc" is not in fact on socketQ, this procedure calls SysErr(soc,ecNoSuchSocket).

Control over assignment of PBIs to sockets is accomplished in a manner that is more complicated to describe than to implement. Associated with each socket are three numbers that determine the maximum number of PBIs that may be assigned to a socket simultaneously. The "total" (soc>>PupSoc.maxTPBI) is the maximum total number of PBIs permitted, while the "input" and "output" values (soc>>PupSoc.maxIPBI. soc>>PupSoc.maxOPBI) determine (independent of the overall total) the maximum number of PBIs that may be assigned for those respective purposes. The "total" maximum prevents a single socket from usurping more than a fixed share of the total PBIs in the system; within that, the "input" and "output" limits, if properly set, prevent all of a socket's allocation from being devoted to packets going in one direction (with resultant potential deadlocks). The "total" allocation must be greater than either "input" or "output", but need not be equal to their sum, since in most applications one expects heavy demands on PBIs in only a single direction.

The actual number of PBIs assigned to a socket at a given moment is reflected in three other cells in the socket: soc>>PupSoc.numTPBI, soc>>PupSoc.numIPBI, and soc>>PupSoc.numOPBI. These are initialized to the corresponding "max" values, decremented whenever a PBI is assigned to the socket, and incremented when the PBI is released. The code responsible for allocating and releasing PBIs (the PupLevel1 background process for input PBIs and the GetPBI procedure for output PBIs) do not permit any of these counts to go below zero; if allocating another PBI would cause a count to be decremented below zero, PupLevel1 will simply discard the Pup and release the PBI, and GetPBI will either block or fail (see below).

The allocations in the socket are also useful when destroying the socket. At the time CloseLevel1Socket is called, there may be PBIs assigned to the socket but that cannot be located at the moment because they reside on some other queue (such as the Ethernet output queue or the pbiTQ). CloseLevel1Socket simply blocks until soc>>PupSoc.numTPBI equals soc>>PupSoc.maxTPBI, at which point it is known that all PBIs have "returned" to the socket and been released.

PBIs may be added to the free pool simply by allocating blocks of size lenPBI and "Enqueue"ing them on pbiFreeQ. One could also remove PBIs from the system by "Dequeue"ing them from pbiFreeQ and freeing them, but of course one has no control over which PBIs are available for release. Note that such changes in the total number of PBIs are not automatically reflected in any socket allocations or in the default allocations contained in dPSIB.

SetAllocation(soc, total, input, output)
Changes the number of PBIs that may be assigned to the socket. "total",

"input", and "output" are the new maximum values. The "total" must be greater than either the "input" or "output". SetAllocation need be called only if the desired allocations differ from the defaults in dPSIB. Alternatively, one may manually change the contents of dPSIB; note that the "num" and "max" values for a given allocation must be the same and that the "total" allocation must be greater than or equal to the "input" and "output" allocations. Changing dPSIB does not affect allocations in sockets that have already been opened. The initial "total" allocation in dPSIB is numPBI-numNets, where numPBI is the argument to InitPupLevel1 that determines the number of PBIs initially created and numNets is the number of directly-connected networks (always one in an Alto). The initial "input" and "output" allocations are each one less than the "total".

GetPBI(soc, returnOnFail [false]) = PBI
Assigns a PBI from pbiFreeQ and charges it to the socket, for output use (that is, it decrements soc>>PupSoc.numTPBI (total) and soc>>PupSoc.numOPBI (output)). If the socket has exhausted its total or output allocation or the pbiFreeQ is empty, then GetPBI blocks unless returnOnFail is true, in which case it returns zero. The PBI returned has its Pup header zeroed so that if the caller later transmits the Pup without setting up source and destination port addresses, the addresses will be correctly defaulted from the socket. The PBI's "queue" pointer is set to pbiTQ, resulting in automatic release of the PBI after it is transmitted. The PBI's "socket" pointer is set to "soc", thereby recording the socket to which it has been assigned.

ReleasePBI(pbi)
Releases the "pbi" and appropriately credits the allocations in the socket to which it was assigned.

CompletePup(pbi, type [], length [])
Causes "pbi" to be completed and transmitted. "Completion" consists of the following operations: "type" and "length", if supplied, are stored in the Pup type and length fields; any zero fields in the Pup source or destination ports are defaulted to the values given in the owning socket's local and foreign port addresses, respectively; the transport control byte (used by gateways) is zeroed; then, if the socket's doChecksum flag is on (the default unless changed explicitly), a software Pup checksum is computed and stored in the Pup. The caller is expected to have set up the Pup's ID, and contents (if any) and its type and length if not supplied in the call. Finally, the PBI is routed to its destination and queued for transmission.

After transmission, the PBI is appended to pbi>>PBI.queue, which (unless changed explicitly by the caller) will be pbiTQ, resulting in automatic release of the PBI. If a different queue is specified for disposal of the PBI (as is done in the BSP package, for example), then the caller is responsible for keeping track of the PBI, and, in particular, for ensuring that all PBIs assigned to the socket have been released before destroying the socket.

A special mechanism exists for broadcasting a Pup on all directly-connected networks. If the allNets bit is set in the PBI status word, then instead of routing the Pup to the destination stated in the Pup header, CompletePup sends the Pup out on each directly-connected network. For each network, the local host address on that network is substituted for the network and host numbers in the Pup source port, and the local network number is also substituted for the destination network field (the checksum is recomputed each time this is done). The "queue" word in the PBI must be pbiTQ (the default) for this feature to work properly.

The allNets mechanism ordinarily causes a Pup to be sent on each directly-connected network, whether or not the network's identity is known. However, if the bypassZeroNet bit is also set, the Pup will not be sent on networks whose identity is not known.

Distribution of received Pups to the correct sockets is the responsibility of a background process called PupLevel1. When a PBI appears on pbiIQ (where it was left by the level 0 input handler), PupLevel1 first performs some checks on the Pup destination address, and discards the PBI if it is not destined for a process in the local host (actually, it enqueues it on gatewayIQ, which, assuming the PupDummyGate module has been loaded, is the same as pbiFreeQ). It then searches the socketQ for a socket whose local socket number matches the Pup destination socket number. If no such socket is found, the PBI is passed to SocketNotFound(pbi), which simply discards the packet (but could be made to do something else by clobbering the SocketNotFound procedure static with a different handling procedure).

Assuming the destination socket is found, PupLevel1 then checks the Pup checksum (assuming the socket's doChecksum flag is on), discarding the PBI if it is incorrect. Finally, the socket's "total" and "input" PBI allocations are checked. If either is exhausted, the PBI is discarded (causing an Error Pup to be returned to the Pup's source); otherwise, the allocations are updated and the PBI is appended to the socket's iQ.

PupLevel1 is also responsible for releasing PBIs on the pbiTQ, which is the default queue to which outgoing packets are appended after transmission.

Another process, GatewayListener, is responsible for dynamically maintaining the routing table pupRT and updating it with information periodically received from gateways. While routing and routing table maintenance are operations performed automatically (by CompletePup and GatewayListener), the format of the routing table is of possible interest to callers in certain cases--for example, in deciding which of several possible remote servers is the best choice in terms of network topology (see the PupNameLookup module for an example of this). The following description is much more than most programmers will wish to know about.

The RT is a hash table object consisting of routing table entries (RTEs) keyed by network number, each containing information about a specific network. If no RTE exists for a particular network, then we know nothing about that network and can't route Pups to it. For a given RTE, if the "hops" field is zero, the network is one to which the local host is directly connected; otherwise, the network may be reached via the gateway whose host number is given in the "host" field (the "hops" field indicates the number of gateways believed to lie along the route to the destination net). In either case, the "ndb" field points to the NDB for the immediate destination network (see "Pup Specifications").

Entry zero in the routing table is special. It refers to a network known to be directly connected to the local host (but whose identity may or may not be known, i.e., we may or may not know its network number). Pups handed to CompletePup for transmission to network zero will be sent over this network. This facility is essential during initialization, before any gateways have been located and the remainder of the RT filled out. It also permits communication between hosts on a network whose identity is unknown due to there being no connected gateways.

The routing table as a whole is treated as an "object", with standard operations defined by a Hash Table Preamble (HTP). The procedures described below are merely renamed versions of the Alto OS's Call0, Call1, etc. The operations return pointers to RTEs, and the caller may operate on the individual RTE by means of ordinary structure references. The defined operations are:

HLookup(rt, net, findFree [...false]) = RTE or 0
 Looks up "net" in the routing table "rt", returning a pointer to the RTE if it is

found and zero if not. If "findFree" is true, then upon failure to find "net" it returns a pointer to a place where an RTE for "net" may be inserted (zero in this case means that the RT is full).

HInsert(rt, net) = RTE
: Inserts an RTE for "net" into "rt", setting the "net" field of the RTE and zeroing the rest of the entry. If an entry already exists for "net", it is overwritten. If no entry already exists, one is created, and if the table is full, SysErr is called.

HDelete(rt, net)
: Deletes the RTE for "net" in "rt", if one exists.

HEnumerate(rt, proc, arg)
: Enumerates all RTEs in "rt", calling proc(rte, arg) for each one.

The following miscellaneous procedures are of possible interest to callers:

PupError(pbi, errorType, string)
: Causes an "Error" Pup to be returned to the sender of "pbi", containing the specified "errorType" and "string". The PBI is released in the process. Consult the "Pup Error Protocol" specification for more information. PupError is called from several places inside PupLevel1 when incoming Pups are rejected for one reason or another.

ExchangePorts(pbi)
: Exchanges the Pup source and destination ports in "pbi". Useful when sending a packet back where it came from (possibly after modifying its contents).

AppendStringToPup(pbi, firstByte, string)
: Appends the supplied "string" to the Pup in "pbi", starting at byte position pbi>>PBI.pup.bytes↑firstByte, then sets the Pup length to include the data so stored. Useful for generating Pups that end in (or consist entirely of) a string, such as Error, Abort, and Interrupt Pups.

SetPupDPort(pbi,port)
: Copies the specified "port" into the Pup destination port field of "pbi".

SetPupSPort(pbi,port)
: Copies the specified "port" into the Pup source port field of "pbi".

SetPupID(pbi,pupID)
: Copies the two words pointed to by "pupID" into the Pup ID field of "pbi".

FlushQueue(queue)
: Dequeues and releases all PBIs presently on "queue".

OnesComplementAdd(a, b)
: Returns the ones-complement sum of "a" and "b".

OnesComplementSubtract(a, b)
: Returns the ones-complement difference between "a" and "b".

LeftCycle(word, count) = result
: Returns the result of left-cycling "word" by "count" mod 16 bits.

MultEq(adr1, adr2, nWords [...2]) = true or false
: Compares the nWords words starting at adr1 with the corresponding words starting at adr2, returning true iff they all match.

Max(a, b); Min(a, b)
Return the arithmetic maximum or minimum, respectively, of "a" and "b". These are treated as signed integers and must differ by less than 2↑15.

DoubleIncrement(adr, offset)
Adds the signed 16-bit integer "offset" to the 32-bit number pointed to by "adr". Note that a negative "offset" will cause the 32-bit number to be decremented.

DoubleDifference(adr1, adr2) = value
Returns as a 16-bit signed integer the result of subtracting the 32-bit number pointed to by "adr2" from the one pointed to by "adr1". If the two numbers differ by more than 2↑15, the result is either 2↑15-1 or -2↑15, depending on the sign of the 32-bit difference.

DoubleSubtract(adr1, adr2)
Subtracts the 32-bit number pointed to by "adr2" from the one pointed to by "adr1", and leaves the result in "adr1".

HHash(logTableSize, key, lvProbe) = increment
Computes a hash probe and increment (for double hashing) from a one-word "key" into a hash table of length 2↑logTableSize. The initial probe is stored in @lvProbe and the increment is returned. "logTableSize" should not be greater than 8.

4. Rendezvous/Termination Protocol Interface

The RTP module (file PupRTP) contains primitives for establishing and breaking connections with foreign processes according to the Rendezvous/Termination Protocol.

The local end of a connection is maintained within the confines of an RTPSoc, an RTP socket structure (defined in PupRTP.decl). This begins with a level 1 Pup socket (PupSoc), but includes the following additional information:

ctx	A pointer to the background context maintaining the connection.
state	The state of the connection (see below).
connID	The connection ID (see "Pup Specifications").
rtpOtherPupProc	A procedure called upon receipt of any Pup that is not part of the Rendezvous/Termination Protocol.
rtpOtherTimer	A timer for use by higher levels of protocol.
rtpOtherTimerProc	A procedure called when rtpOtherTimer expires.

There is some other information (wasListening, rtpTimer) used by the RTP module but not of interest to external programs.

At a given moment, an RTPSoc may be in one of a number of "states". A detailed explanation of the meanings of these states may be found in the memo "Pup Connection State Diagram" (file <Pup>States.Ears).

stateClosed	No connection exists: either none has ever been created or a previously existing connection has terminated.

stateRFCOut	The local process has initiated a request for connection (RFC) to some foreign process. A reply is expected from the remote process.
stateListening	The local process is "listening" for an RFC from any foreign process.
stateOpen	The connection is considered by both parties to have been established. What the cooperating processes do with this connection is a matter of higher-level protocol (e.g., BSP).
stateEndIn	The foreign process has requested that the connection be terminated, and is awaiting a confirmation from the local process.
stateEndOut	The local process has requested that the connection be terminated, and is awaiting a confirmation from the foreign process.
stateDally	A transitory state having to do with the termination handshake (see "Pup Specifications").
stateAbort	The connection has been aborted abnormally by the foreign process.

An RTPSoc is created by calling OpenRTPSocket, which performs various initialization, creates a background process to manage the connection, and interacts with some foreign process in one of three ways (see below) to open a connection. Once the connection is open, the RTP background process monitors the socket for arrival of Pups requesting that the connection be closed or aborted, and updates the state of the socket appropriately. The local process may also request explicitly that the connection be terminated, by calling CloseRTPSocket.

The procedures defined in the RTP module are the following:

OpenRTPSocket(soc, ctxQ [pupCtxQ], openMode [modeInitAndWait], connID [random], otherProc [DefaultOtherPupProc], timeout [defaultTimeout], zone [pupZone]) = true or false

Causes an RTP socket to be created and optional interactions with a foreign process to be initiated. "soc" is a block of length lenRTPSoc which must already have been initialized as a level 1 socket (PupSoc) by a prior call to OpenLevel1Socket. Both the local and foreign port addresses (the "lclPort" and "frnPort" fields in the PupSoc) must be completely established, unless "openMode" is "listenAndWait" or "listenAndReturn", in which case only the local socket number (soc>>PupSoc.lclPort.socket) need be established.

"ctxQ" is a context queue to which a context created by this procedure may be appended. It defaults to pupCtxQ (the "ctxQ" passed to InitPupLevel1).

"openMode" specifies the manner in which the connection is to be opened. If it is "modeInitAndWait", a request for connection to the foreign process is initiated, and OpenRTPSocket then blocks until either the answering RFC is received and the connection's state becomes open (in which case it returns true) or an error occurs (in which case the RTPSoc is closed and OpenRTPSocket returns false). If it is "modeInitAndReturn", the request is initiated in a similar manner, but then OpenRTPSocket returns true immediately and it is the caller's responsibility to monitor the subsequent state of the connection.

If "openMode" is "modeListenAndWait", the socket is placed in a "listening" state.

When a request for connection is received from some foreign process, a reply is generated and the connection becomes open, and OpenRTPSocket returns true. If the mode is "modeListenAndReturn", OpenRTPSocket returns true immediately and it is the caller's responsibility to monitor the subsequent state of the connection.

If "openMode" is "modeImmediateOpen", the socket is immediately placed in the open state (it is assumed that the caller has already performed a rendezvous with the foreign process in some other manner) and OpenRTPSocket returns true.

"connID" is a pointer to a two-word vector specifying the connection ID (see "Pup Specifications"). If not specified, a connection ID is chosen at random. "connID" need never be specified if "openMode" is one of the listening modes.

"otherProc" is a procedure to be called when a non-RTP Pup is received by the socket. This will be described in more detail later. If not specified, "otherProc" defaults to DefaultOtherPupProc, a procedure that simply releases any PBI it is passed (one may change the default by clobbering the DefaultOtherPupProc static with something else).

"timeout" specifies the maximum time OpenRTPSocket will wait (if "openMode" is "modeInitAndWait" or "modeListenAndWait") before timing out and returning false. It (and all other "timeout" arguments in the Pup package) is in units of 10 milliseconds, with a maximum legal value of 2↑15 (a little over 5 minutes), according to the conventions established in the Timer Package. If unspecified, "timeout" defaults to "defaultTimeout", a static defined this module, whose value in the released package is 6000 (i.e., 60 seconds; this is set by the parameter "defaultDefaultTimeout" in PupParams.decl).

"zone" is a free-storage zone from which a context block (of size rtpStackSize) may be allocated. If it is not specified, pupZone (the "zone" passed to InitPupLevel1) is used. Note: OpenRTPSocket calls InitializeContext, so the ContextInit module must be resident (despite what the Context Package writeup says).

CloseRTPSocket(soc, timeout [...defaultTimeout]) = true or false

Requests that the connection rooted in the RTPSoc "soc" be terminated. If "timeout" is nonzero, a normal termination is attempted if possible; if zero (or the attempted normal termination times out), the connection is aborted (terminated abnormally). When the connection has been closed, the context created by OpenRTPSocket is destroyed and returned to the zone from which it was allocated. CloseRTPSocket then returns true if the connection was terminated normally and false if abnormally. The level 1 PupSoc pointed to by "soc" still exists, and it is the caller's responsibility to dispose of it appropriately (generally by calling CloseLevel1Socket).

The process created by OpenRTPSocket (called RTPSocketProcess) has several responsibilities. First, all Pups arriving on the socket's iQ are dequeued and inspected. Ones whose types are part of the Rendezvous/Termination protocol are processed internally. All protocol interactions (including replies, retransmissions, and local state changes) are handled automatically.

Received Pups that are not part of the RTP are passed to the "rtpOtherPupProc" procedure, which is initialized to the "otherProc" argument in OpenRTPSocket. More specifically, the statement

```
(soc>>RTPSoc.rtpOtherPupProc)(pbi)
```

is executed, and it is up to the called procedure to appropriately process and dispose of the PBI. Since this call is made within the context of the RTPSocketProcess, which has only "rtpStackSize" (130 as released) words of stack space, the called procedure cannot make heavy demands on the stack without risk of stack overflow. One might increase rtpStackSize (a static defined in this module, whose initial value is given in PupParams.decl as "defaultRTPStackSize"), but the safest course of action is for the called procedure simply to enqueue the PBI on some queue looked at by another process with more stack space available to it. (One should note, however, that the "rtpOtherPupProc" procedure defined by the BSP module, to be described in the next section, manages to do all its work--a significant amount--without overflowing the RTP process's stack. The main potential pitfall is in calling system procedures such as Ws that require very large amounts of stack space in some cases.)

"Abort" and "Error" Pups, while handled by RTPSocketProcess (for their effects on the socket's state), are also passed on to the "rtpOtherPupProc" procedure, for purposes such as displaying the Pup's text to the user. The RTP module distinguishes between "fatal" and "non-fatal" sub-types of Errors, treating the former the same as an Abort (thereby placing the connection in the "Abort" state) and ignoring the latter; both kinds, however, are passed to "rtpOtherPupProc".

Additionally, the RTPSocketProcess checks for expiration of a timer called "rtpOtherTimer" in the RTPSoc. If it expires, the procedure given in "rtpOtherTimerProc" is called, with the socket as its argument. This facility is used in the BSP module, which also requires the ability to do asynchronous processing. "rtpOtherTimerProc" is initialized to Noop when OpenRTPSocket is called.

The following miscellaneous procedures defined in the RTP module are of possible interest to callers:

RTPFilter(pbi, checkFrnPort, checkID) = true or false
: Does selective filtering of "pbi" against parameters in the socket to which the PBI is assigned, and returns true if the PBI is accepted and false if rejected. First, broadcast Pups (destination host zero) are always rejected. Then, if checkFrnPort is true, the source port address of the PBI is checked for equality with the foreign port address given in the socket. Finally, if checkID is true, the Pup ID in the PBI is checked for equality with the connection ID in the socket.

CompleteRTPPup(pbi, type, length)
: Stores "type" and "length" in the respective fields of the Pup, copies the connection ID from the socket to the Pup, and finally calls CompletePup(pbi) to send it on its way.

5. Byte Stream Protocol Interface

The BSP module (files PupBSPStreams, PupBSPProt, and PupBSPa) contains procedures for sending and receiving error-free, flow-controlled byte streams to and from a foreign process, and for dealing with the other primitives defined by the BSP (namely Marks and Interrupts).

A process's interface to the BSP module is by way of a BSPSoc, a BSP socket structure, which is a further extension of an RTPSoc (which, it will be recalled, is an extension of a PupSoc). The BSPSoc contains a large amount of additional information, most of which fortunately is not of interest to external programs. The items that are of interest are the following:

bspStatus	A word containing various status bits, including the following two:

markPending	A Mark has been encountered while reading the incoming byte stream. Further attempts at input (via Gets or BSPReadBlock) will fail until this bit is cleared (either explicitly or by calling BSPGetMark).
interruptIn	An Interrupt has been received. If the caller depends on this bit for noticing the arrival of Interrupts, then it must clear the bit explicitly after doing so. Interrupts arriving in close succession will not be distinguishable as separate events unless they are intercepted via the "bspOtherPupProc" mechanism, described later.
bspOtherPupProc	A procedure called upon receipt of any Pup not part of the BSP (or RTP).
bspStr	A block containing a BSPStr, a BSP stream structure. This contains the dispatches for interfacing to the operating system's generic stream-handling procedures (Gets, Puts), plus some information specific to the BSP stream.

A BSP stream is created by first opening a connection to a foreign process (by means of the RTP), then calling the following procedure:

CreateBSPStream(soc) = str
Creates and initializes a BSP socket, and returns a pointer to the stream block within it. "soc" must point to a region of length lenBSPSoc, and it must already support one end of an open RTP connection (by having been passed to OpenLevel1Socket and then OpenRTPSocket). If the state of the connection is not stateOpen or stateEndIn, CreateBSPStream returns zero. Otherwise, the stream is completely initialized and the pointer to it is returned. See the sample program at the end of this document for an example of the proper sequence of operations for opening a BSP stream from scratch.

All the generic stream procedures (Gets, Puts, etc.) must be passed "str" as an argument, as should the procedures BSPReadBlock and BSPWriteBlock. However, all other operations on the socket (including specialized BSP functions such as BSPGetMark) must be passed "soc". When necessary, "str" and "soc" may be computed from each other by the following statements:

```
    str = lv soc>>BSPSoc.bspStr
    soc = str-offset BSPSoc.bspStr/16
```

The defined generic stream procedures are as follows. The descriptions of Gets and Puts assume that the default stream error-handling procedure (invoked by Errors(str,ec)) is in use; the real truth appears in the description of Errors.

Gets(str, timeout [...-1]) = byte or -1
Attempts to return the next byte from the BSP stream "str"; returns -1 on any failure. A failure will result if the connection has become closed or a Mark has been encountered in the incoming stream. If "timeout" is -1 (the default), Gets waits indefinitely for data to arrive (or some failure condition to arise); if other than -1, it waits up to "timeout" (units of 10 milliseconds) and then gives the failure return.

Puts(str, byte, timeout [...defaultTimeout]) = true or false
Attempts to output "byte" to the BSP stream "str"; returns true on success and false on failure. A failure will result if the connection has become closed or the operation times out. The "timeout" is defined as for Gets, with -1 meaning wait indefinitely. Note that in general, outputting a byte to a BSP stream

merely causes that byte to be appended to a partially-constructed Pup in memory; only when a Pup is filled up is any packet actually sent over the net. BSPForceOutput (described below) must be called to cause a partially-filled Pup to be closed out and transmitted immediately.

Endofs(str) = true or false
Returns true if there is not presently any data to be read from the BSP stream "str" or a Mark has been encountered. Note that this definition of Endofs is analogous to that for "keys" as opposed to that for disk files; i.e., so long as the connection is still open, Endofs(str) being true says only that there is not now any data to be read, not that there won't be data at some time in the future.

Closes(str) = true or false
Closes the BSP stream "str" and destroys the associated socket, as detailed in the description of CloseBSPSocket (below).

Errors(str, ec) = value
The stream error procedure (which is initialized to BSPErrors by CreateBSPStream) is called under various error conditions arising in Gets and Puts. The error code "ec" will be one of the following:

ecBadStateForGets	Gets has failed because the connection is no longer open.
ecGetsTimeout	Gets has failed because no data became available for reading within the specified timeout.
ecMarkEncountered	Gets has failed because it has encountered a Mark in the stream.
ecBadStateForPuts	Puts has failed because the connection is no longer open.
ecPutsTimeout	Puts has failed because it was not possible to output the byte within the specified timeout.

In each case, the Gets or Puts returns the result of calling Errors with the corresponding error code. The default Errors procedure returns -1 when passed any of the Gets error codes and false when passed one of the Puts error codes, thereby obtaining the failure behavior presented earlier in the descriptions of Gets and Puts.

The remaining procedures operate on a "soc" (BSPSoc) rather than a "str", since they are peculiar to BSP.

CloseBSPSocket(soc, timeout [...defaultTimeout]) = true or false
Closes the connection and destroys the BSPSoc pointed to by "soc". First, if the connection is still in a reasonable state, any pending output is transmitted; CloseBSPSocket will wait up to "timeout" for successful acknowledgment of this data. Next, the connection is terminated by a call to CloseRTPSocket (the description of which includes the interpretation of "timeout"). Then all PBIs still residing on the BSPSoc's various queues are released. Finally, the socket is destroyed by a call to CloseLevel1Socket. The result returned is true if the connection was closed normally, false if abnormally.

BSPGetMark(soc) = byte
Returns the value of the pending Mark byte in the incoming stream, and clears the markPending flag so as to permit future calls to Gets to read data past the Mark in the stream. This procedure will call SysErr(soc,ecBadBSPGetMark) if a Mark has not in fact been encountered.

BSPPutMark(soc, markByte, timeout [...defaultTimeout]) = true or false
  Inserts the specified "markByte" into the outgoing stream. Calling this procedure causes all data up to and including the Mark byte to actually be transmitted immediately. The interpretation of "timeout" and the result returned by the procedure are the same as for Puts.

BSPForceOutput(soc)
  Forces any partially-filled output Pup to be transmitted immediately. This procedure will never block.

BSPPutInterrupt(soc, code, string, timeout [...defaultTimeout]) = true or false
  Generates a BSP Interrupt Pup (see "Pup Specifications") using the specified "code" for the Interrupt Code and "string" for the Interrupt Text. The procedure returns true unless it failed to send the Interrupt due either to the connection no longer being open or to exhausting the specified "timeout".

The BSP module accomplishes much of its work as a result of being given control by the socket's RTPSocketProcess context through two paths: the "rtpOtherPupProc" procedure, called when a non-RTP Pup is encountered, and the "rtpOtherTimerProc" procedure, called when the "rtpOtherTimer" expires. These three cells in the RTPSoc structure are renamed "bspPupProc", "bspTimer", and "bspTimerProc" within the BSP module. By this means, the management of both incoming and outgoing byte streams is accomplished automatically (including the generation of acknowledgments and retransmissions).

Received Pups that are not part of either the RTP or the BSP are handed to the procedure given in the "bspOtherPupProc" cell in the socket. This is initialized to the previous contents of the socket's "rtpOtherPupProc" by CreateBSPStream (which then stores a pointer to the BSP module's own BSPPupProc into the latter cell). The earlier description of "rtpOtherPupProc" (in the section on the RTP module) applies to "bspOtherPupProc".

Received Interrupt packets are also passed to "bspOtherPupProc" after being processed by the BSP module. Note that an Interrupt passed in this manner has been verified to conform to protocol (this is the case also for Abort and Error packets passed up from the RTP module) and may therefore be "believed". Any other type of packet, on the other hand, has had no checking done on it beyond the level 1 interface (where the destination port and checksum were verified).

A note on allocations: this BSP implementation probably will not work at all unless the socket's PBI allocations are at least 3, 2, and 2 for "total", "input", and "output" respectively. High throughput will be gained only by giving the socket somewhat larger allocations (say, 6 to 10 PBIs) for the direction(s) in which high throughput is desired.

In a program with at most one active BSP connection, that socket should be allocated all of the PBIs in the system except one per directly-connected network (there must always be one extra PBI available for receiving incoming packets on each network); this is the default allocation established in dPSIB by InitPupLevel1. In a program with several active connections, one should adjust individual socket allocations appropriately (though probably not simply by dividing the total PBIs by the number of sockets, since doing so typically leads to underutilization of PBIs). Assuming there are plenty of PBIs in the system, it is generally safe to overcommit the system resources (relying on the statistical unlikelihood that all sockets will simultaneously tie up all the PBIs to which they are individually entitled). One should be aware, however, that the higher-level protocols can get into deadlock conditions if the system pbiFreeQ becomes exhausted. For the same reason, a PBI passed to an external program via the "bspOtherPupProc" entry in the socket must be released as quickly as possible, since it is charged against the socket's allocation.

The BSP module includes a static "bspVersion" whose value is (protocol version * 1000) + package version.

6. BSP Block Transfer Procedures

The BSP stream mechanism just presented, while being a "fast stream" in the sense defined by the operating system, is still relatively slow and is therefore not well suited to transferring large volumes of data (such as file transfers between disk and net). A separate module (PupBSPBlock) is provided for accomplishing block transfers at least an order of magnitude faster than by iterated calls on Gets or Puts. This module requires that the AltoByteBlt module (released as a separate package) be loaded as well.

Two procedures are defined in this module:

BSPReadBlock(str, wordP, byteP, count, timeout [...-1]) = count
 Reads a maximum of "count" bytes from the BSP stream "str", storing them in memory starting at byte position "byteP" relative to word address "wordP" (for example, byteP = 0 means the left byte of the word referenced by "wordP"). The transfer terminates under any of the conditions that would cause Gets(soc,timeout) to return -1. The procedure returns the actual number of bytes transferred.

BSPWriteBlock(str, wordP, byteP, count, timeout [...defaultTimeout]) = count
 Writes a maximum of "count" bytes to the BSP stream "str", obtaining them from memory starting at byte position "byteP" relative to word address "wordP". The transfer terminates under any of the conditions that would cause Puts(soc,byte,timeout) to return false. The procedure returns the actual number of bytes transferred.

7. Name Lookup Module

This module (file PupNameLookup) contains a single procedure which will parse a string consisting of any legal inter-network name/address expression and return a Port structure containing that address (suitable for passing to OpenLevel1Socket or plugging into the dPort field of a Pup). See the memo "Naming and Addressing Conventions for Pup" (file <Pup>PupName.Ears) for information on legal expressions.

GetPartner(name, stream [none], port, s1 [...none], s2 [...none]) = true or false
 Parses the BCPL string "name" and stores the resulting address value in the Port structure "port", returning true if successful and false otherwise. "stream", if nonzero, is used for publishing an error message if the conversion is unsuccessful. "s1" and "s2", if supplied, specify the high- and low-order parts of the default socket number, which is substituted into the "port" if the socket number is unspecified in the "name".

 If the "name" consists entirely of address constants (in the form "net#host#socket" or some subset thereof, where the components are octal numbers), then it is parsed locally. Otherwise, GetPartner attempts to establish contact with a Name Lookup server, to which it passes the "name" for evaluation. If the reply consists of several alternative addresses, the "best" one is chosen on the basis of information in the local routing table. Regardless of whether or not the string is an address constant, GetPartner will return false (with the message "Can't get there from here") if no routing table entry exists for the resulting network and several gateway information probes discover no way of reaching that network.

8. Example

The following example program makes use of most of the facilities provided in the Pup package. It is basically a rock-bottom minimal user Telnet (like Chat) with no redeeming features whatsoever.

The main procedure PupExample performs initialization, which consists of creating a large zone, initializing the Pup package, creating a large display window, and creating and starting a context running the procedure TopLevel.

TopLevel first requests the user to type in a foreign port name, which it parses by calling GetPartner (note that the socket number is defaulted to 1, the server Telnet socket). Then a socket is created and a connection is opened. Two new contexts are now created, running the procedures KeysToNet and NetToDsp. TopLevel then blocks until either the connection is no longer open or the second blank key on the right of the keyboard is pressed, at which point it destroys the two contexts it created, closes the connection, and loops back to the beginning.

The KeysToNet procedure blocks waiting for keyboard input, then outputs the typed-in character to the BSP stream and calls BSPForceOutput to force immediate transmission. If the Puts fails, KeysToNet simply blocks forever, in the expectation that TopLevel will detect that the connection is no longer open and take appropriate action.

The NetToDsp procedure blocks waiting for input from the BSP stream. When a normal character is received, it is output to the display. If Gets returns -1, then either a Mark is pending or the connection has ended; if the former, a message is printed and BSPGetMark is called to clear the Mark pending status; if the latter, NetToDsp blocks indefinitely.

```
// PupExample.bcpl

get "Pup.decl"

external
[
InitPupLevel1; OpenLevel1Socket; CloseLevel1Socket; SetAllocation
OpenRTPSocket; CreateBSPStream; GetPartner
BSPForceOutput; BSPGetMark
InitializeContext; CallContextList; Block; Enqueue; Unqueue
InitializeZone; CreateDisplayStream; ShowDisplayStream
Gets; Puts; Closes; Endofs; Ws
keys; dsp
]
static [ ctxQ; myDsp; bspSoc; bspStr ]


let PupExample() be   // initialization
[
let myZone = vec 10000; InitializeZone(myZone,10000)
let q = vec 1; ctxQ = q; ctxQ!0 = 0
InitPupLevel1(myZone,ctxQ,20)
```

```
let v = vec 10000
myDsp = CreateDisplayStream(40,v,10000)
ShowDisplayStream(myDsp)
let v = vec 3000
Enqueue(ctxQ,InitializeContext(v,3000,TopLevel))
CallContextList(ctxQ!0) repeat
]


and TopLevel() be   // top-level process
[
Ws("*nConnect to: ")
let name = vec 127; GetString(name)
if name>>String.length eq 0 then finish
let frnPort = vec lenPort
unless GetPartner(name,dsp,frnPort,0,1) do loop
let v = vec lenBSPSoc; bspSoc = v
OpenLevel1Socket(bspSoc,0,frnPort)
unless OpenRTPSocket(bspSoc,ctxQ) do
    [ Ws("*nFailed to connect"); CloseLevel1Socket(bspSoc); loop]
Ws("*nOpen!")
bspStr = CreateBSPStream(bspSoc)
let keysToNetCtx, netToDspCtx = vec 1000, vec 1000
Enqueue(ctxQ,InitializeContext(keysToNetCtx,1000,KeysToNet))
Enqueue(ctxQ,InitializeContext(netToDspCtx,1000,NetToDsp))
Block() repeatuntil bspSoc>>BSPSoc.state ne stateOpen %
    @#177035 eq #177775  //second blank key pressed
Unqueue(ctxQ,keysToNetCtx); Unqueue(ctxQ,netToDspCtx)
Closes(bspStr)
Ws("*nClosed!")
] repeat


and KeysToNet() be
[
test Puts(bspStr,GetKeys())
    ifso BSPForceOutput(bspSoc)
    ifnot Block() repeat
] repeat


and NetToDsp() be
[
let char = Gets(bspStr)
if char eq -1 then
    test bspSoc>>BSPSoc.markPending
        ifso
            [
            Ws("*nI saw a Mark!")
            BSPGetMark(bspSoc)
            loop
            ]
        ifnot Block() repeat
Puts(myDsp,char)
] repeat


and GetKeys() = valof
[
while Endofs(keys) do Block()
```

```
resultis Gets(keys)
]

and GetString(string) be
[
for i = 1 to 255 do
    [
    let char = GetKeys(); Puts(dsp,char)
    test char eq $*n
        ifnot string>>String.char↑i = char
        ifso [ string>>String.length = i-1; return ]
    ]
]
```

9. Revision History

March 25, 1976

Various minor bugs in both code and documentation were fixed. One serious error in the documentation was in the description of CreateBSPStream, where "lenBSPStr" should have been "lenBSPSoc". The level 1, RTP, and BSP modules each became slightly smaller. Various calls to CallSwat were changed to SysErr with registered error codes.

Level 0: External change: file PupAlEth.bcpl replaced by PupAlEthb.bcpl and PupAlEtha.asm. Internal change: fast (~20-instruction) Ethernet receiver turnaround implemented.

Level 1: External changes: statics pupZone and pupCtxQ added; procedures SetPupDPort, SetPupSPort, SetPupSPort, and FlushQueue added; RT structure definition changed; default pupErrSt is now a "nil" stream rather than "dsp".

RTP: External changes: defaultTimeout and rtpStackSize changed from manifests to statics (with default values defaultDefaultTimeout and defaultRTPStackSize); DefaultOtherPupProc added.

BSP: External change: static bspVersion added. Internal change: the transmission strategy was modified to elicit an acknowledgment before allocation is completely exhausted, hence reducing lost throughput due to round-trip delay.

April 16, 1976

The released package Pup.dm was renamed PupPackage.dm, and a debugging version of the package released as PupDebug.dm. A number of bugs (particularly in level 1) were uncovered while bringing up the software on the Nova.

Level 0: External change: lenPup and lenPBI changed from manifests to statics (defined in level 1) to permit changing PBI size without recompiling the package. Internal change: 100-millisecond transmit timeout and discard added (eliminating deadlocks caused by things like disconnecting the Alto from the Ethernet).

Level 1: External changes: gateway code split out into separate files PupGateway and PupDummyGate, one of which must be loaded (usually the latter); optional extra argument "pupDataBytes" added to InitPupLevel1; default allocations in dPSIB changed to permit a socket to assign all but one of the PBIs in the system;

OpenLevel1Socket defaults the foreign net in some circumstances. Internal change: if "pupDebug" is on, PupLevel1 checks for the pbiFreeQ being exhausted for more than 20 seconds and calls Swat (this usually indicates a deadlock).

BSP: External change: PupBSPb.bcpl replaced by PupBSPStreams.bcpl and PupBSPProt.bcpl (necessitated by Nova BCPL's inability to compile PupBSPb in one gulp).

May 18, 1976

Mostly bug fixes and performance improvements. Some structure definitions were changed, so recompilation of user programs is advised.

Level 0: Internal changes: more assembly code included to reduce packet loss rate; performance statistics gathered if pupDebug on.

Level 1: External change: optional "type" and "length" arguments added to CompletePup.

October 6, 1976

Significant internal changes were made at levels 0 and 1, and several new capabilities were added. However, for the most part the changes are upward-compatible. Many structure declarations changed, so recompilation of programs that "get" any Pup .decl files is required.

Level 0: External changes: SendEtherPup removed; EncapsulateEtherPup and SendEtherPacket added; ability to send and receive non-Pups implemented.

Level 1: External changes: PupRoute file added; PupGateway module deleted from public Pup package release; routing table completely reorganized; new procedures HLookup, HInsert, HDelete, HEnumerate, HHash added; pupErrSt removed; mechanism added for broadcasting to all connected networks; procedures DoubleIncrement, DoubleDifference, Double subtract included (formerly in BSP module). Internal change: GatewayListener dynamically maintains the best path to each network and purges RTEs of networks for which no routing information has been received recently.

BSP: Internal change: adaptive retransmission timeout implemented to reduce packet loss rate when sending through slow networks or to slow destinations (e.g., Maxc).

March 21, 1977

Mostly bug fixes. Some structure definitions at level 0 were changed, so recompilation of user programs is advised.

Level 0: SendStats operation added to the NDB object.

July 11, 1977

No external changes. The Ethernet driver was rewritten to eliminate several low-probability race conditions and improve performance slightly. The driver now uses the "input under output" feature unconditionally, so problems may be encountered on Alto-Is running old microcode.

Queue Package

This package implements a simple set of queue primitives. They are written in assembly language, so they are small (the entire package is 69 instructions) and fast (see timings).

All the procedures are contained in AltoQueue.br, which is assembled from AltoQueue.asm. A Nova version of this package is available.

All queue primitives make use of two structures: the Queue header (hereafter abbreviated Q) and the Item.

```
structure Q: [
          Head word        // Pointer to first Item on Q
          Tail word        // Pointer to last Item on Q
          ]

structure Item: [
          Link word        // Link to next Item
          Remainder word whatever
          ]
```

An empty queue is denoted by Q.Head equal to zero and Q.Tail unspecified. The last Item on a queue has zero in its Link field. An Item either passed to or returned from the following procedures may have an arbitrary Link word. The Q and Item parameters in these procedures are of course pointers to the respective objects.

Enqueue(Q,Item)
Appends the Item to the Q, thereby making it be the tail item. Enqueue will call Swat if Item is zero (which is a common source of bugs).

Dequeue(Q) = head Item or zero
Removes and returns an Item from the head of the Q, or zero if the Q is empty.

InsertBefore(Q,Successor,Item) = true or false
Inserts the Item in a specific place on the Q, immediately before the specified Successor item. Returns true normally, false if Successor was not found on the Q.

InsertAfter(Q,Predecessor,Item) = true
Inserts the Item in a specific place on the Q, immediately after the specified Predecessor item. Returns true always (undefined things will happen if Predecessor is not actually on the Q).

Unqueue(Q,Item) = true or false
Removes a specific Item from the Q. Returns true normally, false if Item was not found on the Q.

QueueLength(Q) = integer
Returns the number of items on the Q.

All the queue routines are completely race-free, and both interrupt and non-interrupt code may safely access the same Q simultaneously. However, calls to these procedures must be made with interrupts enabled, since they execute "dir" and "eir" internally for race avoidance.

Timings for these procedures are now given. These counts are simply the number of instructions executed, not including the instruction that called the procedure. The procedures InsertBefore, Unqueue, and QueueLength must search the queue from its head until they reach Successor, Item, or the end of the queue respectively; the factor "n" in the timings is the number of items looked at.

Enqueue	14 if Q previously empty
	13 otherwise
Dequeue	10 if Q empty
	11 otherwise
InsertBefore	10+4n
InsertAfter	15 if Predecessor was previously the tail
	14 otherwise
Unqueue	12+4n if Item was previously the tail
	11+4n otherwise
QueueLength	6+4n

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

ReadUserCmItem

A package is now available for reading items from user profile files such as User.Cm. This package provides one procedure:

ReadUserCmItem(stream, string)

where stream must be a standard Alto stream which delivers characters from User.Cm (or any other file in the same format), and string must be a pointer to a 128-word buffer area. ReadUserCmItem reads the next item from the stream and stores it in the buffer area in the form of a standard Bcpl string. ReadUserCmItem returns a value which identifies what type of item was read:

$E (end)	End of stream. String is meaningless.
$N (name)	The item was of the form [string].
$S (string)	The item was of the form "string".
$L (label)	The item was of the form string:.
$P (parameter)	The item was a line not conforming to any of the above (terminated by <cr>).

For items of types $L and $P, ReadUserCmItem removes initial blanks and tabs if any. Blank lines are skipped. If an item will not fit in a Bcpl string (i.e. is longer than 255 characters), characters beyond the 255th are simply discarded.

Here is an example file with the list of values and strings returned by ReadUserCmItem.

File:

```
[BRAVO]
LEAD: Line lead = 6, Paragraph lead = 12

[DDS]
Selspec: "D*"
```

Values and strings returned by successive calls of ReadUserCmItem:

```
$N      BRAVO
$L      LEAD
$P      Line lead = 6, Paragraph lead = 12
$N      DDS
$L      Selspec
$S      D*
$E
```

RenameFile

This package contains a single procedure, RenameFile, which changes the name of a file in an Alto file system. The procedure handles multiple directories and versions, changes the file's serial number to invalidate old hints, updates leader page information, logs its action in the system log, works with BFS or TFS, and generally tries to do the job as throughly as if it were part of the Alto OS directory module. RenameFile only works in Operating System versions 13 or later (in earlier versions it returns false without doing anything).

RenameFile(oldName, newName, versionControl [verLatest], errRtn [SysErr], zone [sysZone], logInfo [0], disk [sysDisk]) = true or false
Deletes the directory entry 'oldName' (after applying versionControl), changes the file serial number, and creates a directory entry 'newName', returning true if successful. 'OldName' must exist and 'newName' must not exist (unless versions are enabled in which case the next version of 'newName' is created). RenameFile will call errRtn(ecZoneTooSmall) if there is not enough space in zone to allocate a page-sized buffer.

Ring Buffer Routines

This package consists of a set of fairly fast assembly-language procedures for buffering data by means of circular buffers. The package comes in two versions: a "byte" version (RINGBYTES.ASM) that deals with bytes and packs them two per word, and a "word" version (RINGWORDS.ASM) that deals with full words. The procedures in the two packages are called identically, so one may substitute the "word" version for the "byte" version to gain about a factor of two in speed at the cost of using buffer space only half as efficiently. Source and binary files for the two versions are supplied in the file RINGBUFFER.DM. A Nova version of this package is available.

A ring buffer is described by a Ring Buffer Descriptor (RBD), which is the address of a 4-word patch of memory provided by the user, initialized through a call to InitRingBuffer, and thereafter maintained by the routines in the package. The "byte" and "word" versions of the routines make different uses of the RBD, but this is of no interest to callers.

InitRingBuffer(RBD,Buffer,Length)
    Initializes the RBD to describe a block of storage starting at "Buffer" and of length "Length" (in words).

ResetRingBuffer(RBD)
    Renders the ring buffer described by RBD empty.

RingBufferEmpty(RBD) = true or false
    Returns true if the buffer is empty.

RingBufferFull(RBD) = true or false
    Returns true if the buffer is full.

ReadRingBuffer(RBD) = Item (byte or word)
    Returns the next Item in the ring buffer if there is one, or -1 if there isn't. Obviously, if the "word" version of the package is being used and -1 is a possible Item, then the caller should check with RingBufferEmpty before calling ReadRingBuffer.

WriteRingBuffer(RBD,Item) = true or false
    Attempts to put Item into the ring buffer and returns true if successful. The "byte" version of this procedure depends on the left half of Item being zero.

When these routines are used to pass streams of data between interrupt-level and non-interrupt-level code, the following precautions should be observed to avoid races:

1. For a given RBD, neither ReadRingBuffer nor WriteRingBuffer should be called both from interrupt level and from non-interrupt level. However, ReadRingBuffer may be called from interrupt level and WriteRingBuffer from non-interrupt level or vice versa.

2. InitRingBuffer and ResetRingBuffer should not be called from interrupt level.

3. Calls to all routines should be made with interrupts on, since some of them execute "dir" and "eir" internally. (This is not a problem if the BCPL Interrupt Package is being used.)

The following information is provided for debugging purposes only, and one should not write code that depends on it.

The "byte" version of the package lays out the RBD in the following way:

```
structure RBD: [
        Begin word          // Pointer to start of buffer
        Length word         // Buffer size in bytes
        Read word           // Current read index
        Write word          // Current write index
        ]
```

The buffer is treated as an array of bytes, packed left to right and indexed starting at zero. The Read and Write indices refer to the last byte read or written.

The "word" version of the package uses the RBD in this way:

```
structure RBD: [
        Begin word          // Pointer to start of buffer
        End word            // Pointer past end of buffer
        Read word           // Current read pointer
        Write word          // Current write pointer
        ]
```

The End word points to the first word beyond the end of the buffer; i.e. its value is Begin plus the length of the buffer. The Read and Write pointers point to the next word to be read or written.

Rough timings for the important procedures are now given. The counts are simply number of instructions executed, not including the instruction that called the procedure.

	"byte"	"word"
RingBufferEmpty	9	9
RingBufferFull	10	11
ReadRingBuffer	20.5 normally	12 normally
	9 if empty	9 if empty
WriteRingBuffer	25 normally	13 normally
	13 if full	13 if full

RWREG - procedures for reading and writing microprocessor memories

Procedures are now available for reading and writing the Alto microprocessor memories (R/S, constant, microinstruction) under program control. These procedures are of greatest use when debugging new microcode, but may also be useful in conjunction with language emulators such as Lisp and Mesa.

For the purposes of this package, the R registers are numbered 0 through 37b, and the S registers from 41b through 77b (register 40B is the M register).

ReadReg(regno) -> value

Returns the contents of register regno.

WriteReg(regno, value)

Writes value into register regno.

MakeXregDesc(regno, flag) -> desc

Returns a "register transfer descriptor" which contains an encoding of the register number regno and the operation specified by flag (false means read, true means write).

DoXreg(desc, value) -> value

Performs the operation specified by the register transfer descriptor desc, returning the contents of the register if a read, or writing value into the register if a write.

The reason for MakeXregDesc and DoXreg is that "compiling" the descriptor in advance allows the actual transfer to be done more quickly.

ReadConReg(conno) -> value

Returns the contents of constant memory location conno.

ReadInsReg(loc, v2)

V2 must be a pointer to a 2-word area. Reads the contents of microinstruction RAM location loc into v2!0 and v2!1. Note that ReadInsReg is not capable of reading the microinstruction ROM, only the RAM.

WriteInsReg(loc, v2)

Writes v2!0 and v2!1 into microinstruction RAM location loc.

***** NOTE TO RAM PROGRAMMERS: RWREG uses RAM locations 1770 through 1772.

ScanFile - a package for rapid sequential file scanning

This package enables a program to scan Alto files at full disk speed, including overlapping disk transfers with computation. The package is written entirely in Bcpl and uses only standard OS facilities.

To initialize the package, call
ScanFile(fp, bufferAddress, bufferSize[, fa, disk])
where fp is a file pointer as described in the Alto OS manual and bufferAddress is the beginning of a block of bufferSize words. If fa is given, it must be a file address as described in the Alto OS manual, and scanning will begin with the file page specified by fa. The disk address in fa must be correct, not just a hint. If disk is given, it must point to a disk descriptor as described on p. 52 of the Alto OS manual; otherwise, ScanFile uses sysDisk, the standard system disk.

ScanFile returns an instance pointer (ip) which points to a structure ScanFile sets up in the buffer area. The minimum size for the buffer area is available in a static called
ScanFileFixedSize
and each additional page (400b-word) buffer requires
ScanFileBufferSize
words.

To get the next page of the file, call
ScanBuffer(ip, fa)
where ip is the instance pointer returned by ScanFile and fa is a pointer to a file address structure as described in the Alto OS manual. If the end of the file has not been reached yet, ScanBuffer returns the address of a page buffer containing the next page of data, and fills in the fa with the page number, disk address, and number of characters of data in the page. If the end of the file has been reached, ScanBuffer returns 0. Note that the contents of a page buffer are only guaranteed valid until the next call on ScanBuffer. Note also that the first page delivered by ScanBuffer is the first page of data, not the leader page.

When you are finished scanning a file, call
ScanFinish(ip)
where ip is the instance pointer. If you don't do this, the next use of the Bfs (e.g. by the OS) may throw you into Swat.

It is possible, although not particularly recommended because of arm movement, to scan more than one file simultaneously with ScanFile. Of course, each file being scanned requires a separate call on ScanFile and its own buffer area.

ScanFile currently only handles the standard Alto Diablo disks (model 31 or 44), not Tridents. If the need arises, ScanFile can be extended to handle Tridents fairly easily.

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 description, 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.

SDialog -- Simple Dialoging Package

SDialog is a package of BCPL subroutines that will aid a program in carrying on a teletype style interaction with its users. Here is a list of its features:

1) SDialog handles all the display and keyboard I/O, including such things as backspacing over a character.

2) SDialog handles converting things between their representations as strings and their internal form.

3) There is help provided when the user types in an illegal or malformed response.

4) There are provisions for defaulting the user responses.

5) SDialog is small (it's probably fast too, but that doesn't matter).

Before proceeding any further you should read the memo entitled "Users' Guide for 'Simple Dialoging'" in <Parsley>SDlg.ears. The rest of this discussion will assume a familiarity with that memo.

SDialog will handle dialog about several different kinds of things. Each of these things is assigned a "radix". Note that as is usual in BCPL, all "values" are always 16 bits, but some of those values may really be pointers to (addresses of) multiword vectors. Here is a list of the legal radices (the declarations may be found in the file UtilStr.d):

integers (>=2) -- Only radices of 2, 8, 10, and 16 will really work right. When integers of radix 2, 8, and 16 are shown to the user, they are always considered unsigned.

radixString (0) -- a normal BCPL string

radixFileName (-3) -- a BCPL string, but user responses are checked for legality

radixCharCode (-1) -- the ASCII code of a character, i.e., 0 <= value <= #377

radixSwitch (-2) -- the value is either TRUE or FALSE

If you wish to do dialoging about something other than the above, then you should tell SDialog that you are dialoging about a radixString and then convert the users response to your internal form yourself.

Here are some notational conventions for what follows: Arguments enclosed in square brackets are optional. If an optional argument is followed by a slash, then whatever follows the slash is the default value for that argument. If there is no slash, then there is no default value. Whatever follows "->" is an indication of the return value of the routine (if any).

There is one basic procedure:

```
Dlg (prompt, radix, [defaultValue, [pointer, [defaultExtension] ] ])
   -> value
```

where prompt is a string, radix is one of the list above, defaultValue and value are "values" of that radix, pointer is just that, and defaultExtension is a string. pointer is where to put the (converted) response if a value to this radix is really a pointer, e.g., if radix is radixString.

Since this routine would be somewhat awkward to use, there are several other routines that call it. In general there are two routines per radix, one that takes a default value and another that doesn't.

```
    DlgNum (prompt, [radix/10]) -> integer
    DDlgNum (prompt, defaultNumber, [radix/10]) -> integer

    DlgStr (prompt, resultString)
    DDlgStr (prompt, defaultString, resultString)

    DlgFileName (prompt, resultFileName, [defaultExtension])
        -> resultFileName>>SL
    DDlgFileName (prompt, defaultFileName, resultFileName,
        [defaultExtension])

    DlgSw (prompt) -> Switch
    DDlgSw (prompt, defaultSwitch) -> Switch

    DlgChar (prompt) -> CharCode
    DDlgChar (prompt, defaultCharCode) -> CharCode

    DlgCA (prompt)
```

DlgCA is what you should call when you want something confirmed, but don't want any "value". DlgCA merely waits for the user to type one character. If it's a positive response it returns. If it's negative it calls DlgErr (see below).

No problems are occasioned by having defaultString and resultString be the same (this holds for file names too). In the dialoging about file names it's possible to specify a default extension for that file name with or without a default file name. The default extension will be added to the user response if and only if that response did not include a period. string>>SL means the length of the string.

Now will talk about dialoging errors. Whenever anybody discovers an error in a user response, he should call

```
    DlgErr ([msg1, [msg2, [errLoc, [errStack] ] ] ])
```

where msg1 and msg2 are strings (or 0), errLoc is the label where control is to go, and errStack is the value that should be in the stack pointer (address of a frame) when control gets to errLoc. DlgErr types the messages to the user followed by a carriage return and does a GotoLabel (errStk, errLoc, nil). Note that errLoc and errStack had better go together.

Actually things are a bit better than this. There is a routine

```
    DlgInit ([errLoc, [inStream, [outStream] ])
```

that may be used to set errLoc and errStack. errLoc is generally set explicitly using DlgInit and errStack is set to the frame of the caller of DlgInit. The idea is that just before you're about to get a parameter from the user that he/she might screw up on, call DlgInit with a label that is just before the call on some dialoging routine. Then if an error is discovered, call DlgErr with the appropriate error message. The error message will appear and the user will get another chance to type in the parameter. There are examples of this sort of usage of DlgInit and

DlgErr in the source code files for the subsystems IcSEM and IcGerb. Here is an example:

```
    let inFileName = vec lFileName
    DlgInit (NoInFile)
  NoInFile:
    DlgFileName ("Input", inFileName, "icarus")
    let inS = OpenFile (inFileName, ksTypeReadOnly)
    if inS eq 0 do DlgErr (inFileName, " doesn't exist")
    if Gets (inS) ne icarusPassWord do
        DlgErr (inFileName, " isn't an Icarus file -- wrong password")
```

The reason why DlgInit ought to be used (rather than DlgErr alone) is that SDialog itself sometimes calls DlgErr and errLoc and errStack should be correct before that happens. SDialog checks user responses for such things as: no letters or illegal digits in integers, only legal characters in file names. If SDialog sees such an inappropriate response from the user, it calls DlgErr, so things ought to be set up so that the user gets to try again on his response, and that's what DlgInit does for you.

There are three "global" variables in SDialog that a user program may change: dlgDefaulted, dlgInS, and dlgOutS. The latter two are streams. They default to keys and dsp respectively. Feel free to set up your own display or file streams. Note that these globals get set every time DlgInit is called.

The global variable dlgDefaulted is a boolean. It says whether or not the user has asked to take the defaults for the rest of the dialog. Some strange programs may want to intervene in this.

There are two more routines that are available (but probably no one will want to use them):

```
    DlgGetParameter (string, [defaultSwitch])
    DlgBackaChar (char)
```

DlgGetParameter does all the work of Dlg after the prompt has been displayed and up to the conversion of the response, i.e., it displays the default response (if any) and receives the user's response (with echoing). DlgBackaChar will backspace over and erase a character on the display.

SDialog uses several routines from the package UtilStr, so normally SDialog and UtilStr should be loaded together. You may want to combine and tailor the source code of these two packages for your own uses. Help is available from the maintainer(s) of the packages.

Cubic spline packages: SPLINE1 & SPLINE2

The files SPLINE1.BCPL and SPLINE2.BCPL contain procedures for fitting cubic splines to sets of data points, called knots. The algorithms are documented in the report "Spline Curve Techniques" (by Baudelaire, Flegal, & Sproull), May 1977.

The two packages contain a procedure of the SAME name, with an IDENTICAL calling sequence:

success←ParametricSpline(N, x, y, p1x, p2x, p3x, p1y, p2y, p3y, type [0])

N n=|N| is the number of knots. The sign of N tells whether the knot coordinates are given in integer format (N is negative) or floating point format (N is positive).

x, yare two tables containing the coordinates of the knots. They are of length n (integer) or 2*n (floating point).

p1x, p2x, p3x, p1y, p2y, p3y are six tables of length 2*n in which the coefficients defining the parametric splines are returned (floating point). These coefficients are, respectively, the first, second and third derivatives at each knot of the cubic splines x(t) and y(t), t varying between 0 and 1. Notice that, although only the first n-1 values of these derivatives are necessary, the arrays should be of length 2*n.

typės either 0 (for natural end conditions, i.e. open ended curve) or 1 (for periodicity, i.e. cyclic curve). In the later case, it is mandatory that the first and last knots be identical. The type defaults to 0.

The implementation of the parametric spline algorithm is different in the two packages: SPLINE1 implements a unit step parametrization (algorithm 1.2.7), while SPLINE2 implements a chord length parametrization (algorithm 1.2.5).

In addition, SPLINE2 contains the procedure CubicSpline which computes a general non-parametric cubic spline (algorithm 1.2.5). The calling sequence is:

success←CubicSpline(N, x, y, p1y, p2y, p3y, type [0])

with the same conventions as above.

All the procedures need free storage, which they get from a zone you must provide by setting the static PSzone. The amount of storage needed is as follows: In the basic case (n positive, type=0): enough for 8 floating point registers (16), plus 4*n. If n is negative, the coordinates have to be converted to floating point format: so add 4*n. If type is 1, add 6*n.

The static PSerror points to an error procedure that simply traps to SWAT. The error routine is called by the statement: "... resultis PSerror(errorNumber);" You may substitute your own error handling routine. errorNumber=1 means "not enough storage." Other errors are probably fatal.

The spline packages use the FLOAT package for all arithmetic calculations. The format of floating point numbers is consistent with the conventions of that package.

Strings Package

This package provides a small set of useful string-manipulation primitives. There are two independent modules: a "streams" module implementing standard stream operations reading and writing strings, and a "utility" module containing a small set of procedures for concatenating, extracting, and comparing strings.

The utility operations parallel some of those provided in Bruce Parsley's UtilStr package. The principal departures from that package are:

1. Procedures that create new strings get storage by allocating it from sysZone rather than requiring that the caller supply it.
2. Operations on large strings are relatively efficient because the ByteBlt package is used.
3. No format conversion operations are provided, since the availability of string streams makes it possible to use existing software that formats output to streams (e.g., the procedures in the operating system or the Template package).

The .br files are packaged as Strings.dm, and the sources are contained in StringsSource.dm, which also includes various command files.

1. String Streams

The "streams" module (file StringStreams.br) provides one external procedure for creating a string stream; all other access to the stream is via the standard stream operations. The package makes use of the operating system's "fast streams" mechanism, so it is relatively efficient when dealing with long strings.

CreateStringStream(string, maxLength [0], firstChar [1], zone [sysZone]) = ss
  Creates and returns a string stream reading or writing the specified BCPL string. If maxLength is zero (the default), assumes that an existing string has been supplied (presumably for reading); if nonzero, assumes only that a block of storage capable of holding a string of maxLength characters has been provided. firstChar is the index of the first character to be read or written (remember that the first character of a BCPL string is numbered 1, not 0). By appropriate setting of maxLength and firstChar one may read partial substrings or append to existing strings. The stream structure is allocated from the specified zone.

Gets(ss), Puts(ss, c)
  Reads or writes the next character in the string. If the end of the string is exceeded (either its existing length or maxLength), Errors(ss, ecEof) is called (ecEof = 1302).

Endofs(ss) = true or false
  Returns true if the next Gets or Puts would call Errors.

Closes(ss)
  If any Puts operations have been executed, updates the string's length to be the current position (i.e., the index of the last character read or written). Then destroys the stream by returning it to the zone from which it was allocated.

An additional module StringOEP.br is provided. It declares the Overlay Entry Points (OEPs) for the StringStreams module, which need be done only if the module is loaded as part of an overlay. Consult the author for further information.

2. String Utilities

The "utilities" module (file StringUtil.br) requires that the ByteBlt package (file AltoByteBlt.br) also be loaded. All strings created by these procedures are allocated from a zone (default sysZone), so the caller should return them by calling Free when done with them.

ExtractSubstring(string, first [1], last [string>>String.length], zone [sysZone]) = newString
    Extracts the "first" through "last" characters of the supplied string and returns the result as a new string. The defaults are such that the entire source string is copied, thereby providing a convenient way to create copies of strings.

ConcatenateStrings(s1, s2, free1 [false], free2 [false], zone [sysZone]) = newString
    Returns the result of concatenating strings s1 and s2. Then frees s1 if free1 is true and s2 if free2 is true. This facilitates writing embedded string expressions whose result is a single string, with all intermediate strings discarded. (All strings must belong to the same zone.)

CopyString(dest, source)
    Simply copies the source string into the block pointed to by dest, which had better be big enough. This procedure does not allocate new storage.

StringCompare(s1, s2, first1 [1], last1 [s1>>String.length], first2 [1], last2 [s2>>String.length]) = result
    Compares the first1 through last1 characters of string s1 with the first2 through last2 characters of s2. Returns a code describing the outcome:

-2	s1 is an initial substring of s2.
-1	s1 is "less than" s2 but not an initial substring.
0	s1 is "equal to" s2.
1	s1 is "greater than" s2.

    Lower-case letters collate with their upper-case equivalents. The arguments beyond s2 are optional and default to the entire respective strings.

3. Revision History

May 24, 1977

First release.

July 8, 1977

Optional zone argument added to ExtractSubstring and ConcatenateStrings.

Template Package

The Template Package contains a single procedure, PutTemplate, which formats output to a stream according to a template provided as a string. This software serves essentially the same purpose as the existing Format Package, but is implemented much more efficiently (it contains one-third as much code, requires one-fifth as much stack space, and runs over ten times as fast as Format). The major difference from Format is that PutTemplate outputs to a stream rather than to a string (though of course one could obtain the same effect by outputting to a string stream). The template syntax is also different, and PutTemplate omits a few miscellaneous capabilities such as hexadecimal output. A Nova version of this package is available.

PutTemplate(stream, template, par1, par2, ..., parN)
Writes the "template" (a BCPL string) to "stream". Within the template may appear zero or more escape sequences of the form:

$ modifiers command

For each of these, the next parameter (starting at "par1") is substituted, with conversion as specified by the escape sequence.

An escape sequence consists of a dollar sign, followed by an optional modifier sequence, followed by a one- or two-letter command (upper and lower case are equivalent). There should not be any spaces or other extraneous characters within the escape sequence. A dollar sign may be included literally in the template by writing "$$".

The defined escape sequences are as follows. "#" stands for the optional modifier sequence (to be explained shortly).

$S	Treat the parameter as a BCPL string.
$US	Treat the parameter as an unpacked string. This is a vector consisting of a character count in the first word followed by that number of characters right-justified in succeeding words.
$C	Treat the parameter as a single right-justified character.
$#D	Output the parameter as a decimal integer.
$#O	Output the parameter as an octal integer.
$#B	Output the parameter as a binary integer.
$P	Treat the parameter as a procedure, passing it the stream and the next parameter as arguments (hence a $P uses up two of PutTemplate's parameters).

In the case of numeric output commands (namely $D, $O, and $B), a modifier sequence may be included between the dollar sign and the command. These modifiers further control the interpretation and formatting of the output.

One kind of modifier is a decimal number (of one or more digits). If present, it specifies the minimum field width to be used in outputting the number. If the number contains fewer digits than specified for the field width, then leading fill characters (normally spaces; see below) are supplied. However, if the number contains more digits than will fit in the field, the width specification is ignored and as many digits as necessary are printed. The default field width is one.

Other modifiers consist of single letters and are as follows:

U	Treat the parameter as an unsigned rather than a signed integer. (Generally one should invoke this modifier when outputting numbers in octal or binary.)
E	Treat the parameter as a double-precision (32-bit) integer (mnemonic "Extended"). In this case, the argument is a pointer to a two-word vector containing the integer to be printed, with the high-order 16 bits in the first word and the low-order 16 bits in the second. Double-precision numbers may be treated as either signed or unsigned.
Fx	Use the character "x" for leading fill, when necessary, rather than space.

For example, the escape sequence "$12UEF0O" will output an unsigned, double-precision octal number, right-justified in a 12-digit field, with leading zeroes printed as zeroes rather than spaces.

PutTemplate will call SysErr if it encounters an escape sequence it doesn't understand or if there aren't enough parameters to fill all the escape sequences in the template. PutTemplate can handle a maximum of 20 parameters.

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.

Timer Package

This package contains a small set of trivial procedures for setting, testing, and blocking on timers. It exists as a separate package so as to isolate its Alto-dependent implementation in one place (an exactly compatible version for the Nova is also available). For example, calls to this timer package are scattered throughout a rather large body of new Alto Pup software which is intended to run without change on the Nova as well. The package is written in assembly language and contains only 33 words of code.

A "Timer", as used in this package, is a single word whose address is passed to the procedures in this package and used as a temporary variable by those procedures. The actual manner in which this word is used is not of interest to callers.

The unit of time is 10 milliseconds (again, for compatibility with the Nova). Since the Alto clock used in this package (memory location #430) has an period of 39 milliseconds, intervals passed to these procedures must be converted to Alto clock units. Fractions of an Alto tick are rounded up, with the effect that the actual elapsed time will be at least as great as that specified, possibly as much as 39 milliseconds greater. These procedures are not intended for use in making precise measurements or maintaining clocks, but rather for controlling asynchronous operations such as Pup timeouts and retransmissions.

InitializeTimer()
    Initializes the timer package. It should be called once at the beginning of a program that uses the other routines in this package. In the Alto version, InitializeTimer is a complete no-op, and is included only for compatibility with the Nova version in which some initialization is actually required.

SetTimer(lvTimer,Delta)
    Sets the timer word pointed to by lvTimer so that it will expire at the current time plus Delta (in units of 10 milliseconds). Delta must be less than 2↑15 (a little over 5 minutes).

TimerHasExpired(lvTimer) = true or false
    Returns true if the timer pointed to by lvTimer has expired (i.e., the interval Delta specified in the last SetTimer has elapsed).

Dismiss(Delta)
    Blocks (i.e., suspends execution) until the interval Delta has elapsed (Delta is specified in units of 10 milliseconds and must be less than 2↑15). Blocking is accomplished by calling the external procedure Block(), which is defined in the BCPL Context Package and causes control to pass to other processes. If the Context Package is not being used, it suffices to define an external procedure Block() which just returns immediately. The effect of Dismiss(Delta) is approximately equivalent to the following BCPL code, but implemented somewhat more efficiently:

```
let Timer=nil
SetTimer(lv Timer,Delta)
until TimerHasExpired(lv Timer) do Block()
```

Bcpl/Asm procedure tracing package

This package makes it possible to trace Bcpl and Asm procedures on the Alto, similar to the TRACE facility available in Interlisp. The package normally uses Taft's Template (formatted output) package, but is usable without it.

To start tracing calls and returns of procedure proc, call

```
        Trace(proc, str)
```

where str, as described below, specifies the format of the output which Trace produces. To stop tracing proc, call

```
        UnTrace(proc).
```

If you want to trace a procedure but produce all the output yourself, you can call

```
        ProcTrace(proc, tproc)
```

which turns on tracing of proc, but instead of using the second argument as an output template, causes tproc to be called just before proc is entered and just after proc returns. The call when proc is entered is of the form

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

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

```
        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 package or the PutTemplate procedure), 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
```

or to the system display stream dsp if TraceStream is zero. If you set the static

```
        TraceLines
```

to a non-zero value, the tracing routines will pause after every TraceLines lines of output, as follows:

```
        print 3 *'s,
        wait for a character to be typed,
        print 2 more *'s,
```

and then proceed. Other output to the same stream (e.g. from the program being traced) will not be counted in the line count, since the tracing routines have no way to intercept it, but the package constructs a stream

```
        TraceOuts
```

to which you can do Puts and which does the line counting.

The output produced for a Trace'd 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

```
        TraceIndent(stream)
```

writes the appropriate number of spaces on a stream, e.g. TraceOuts.) 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 = $O"). In this case, the portion of str before the $; is used as the template given to PutTemplate 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 a variable number 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 $6UO"), 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 Template package; 3 and 4 do require Template if str contains any $ fields. In the latter case, if the Template package is not loaded, all values will be printed with Wos. Use of ProcTrace does not require the Template package, unless, of course, the user's own trace-print procedures use Template.

Note that Trace can be called from Swat, but only with str omitted or zero. ProcTrace and UnTrace can be called freely from Swat.

UtilStr -- Utility and String Package

I. Introduction

UtilStr is a collection of BCPL subroutines that do string manipulation, double precision arithmetic, and some other things.

It should be noted that these routines don't have much to do with each other, so if you only want to use some of them, feel free to extract or copy from the source code. UtilStr uses definitions from the file UtilStr.d. If you use UtilStr in some program, you will probably want to do a "get" on this file. UtilStr only uses routines from the O.S.

There are three sections to this document. The rest of this Introduction will give the various notational and naming conventions used in the other two. Section II, "Descriptions of Subroutines", gives the calling sequences and a brief description of each routine. Section III, "List of Subroutines", just lists all the calling sequences. It is meant to be used for quick reference purposes.

Here are some notational conventions for what follows: Arguments enclosed in square brackets are optional. If an optional argument is followed by a slash, then whatever follows the slash is the default value for that argument. If there is no slash, then there is no default value. Whatever follows "->" is an indication of the return value of the routine (if any). str>>SL means the length of a string.

Here is a list of conventions for argument names. In general, the "type" of an argument is indicated by its name.

```
    value -- a value is always associated with a radix which
        specifies the value's type
    radix -- one of the following constants
        (manifests are defined in the file UtilStr.d):
        2 -- binary integer
        8 -- octal integer
        10 -- decimal integer
        16 -- hexadecimal integer
        radixString (0) -- a BCPL string
        radixFileName (-3) -- a BCPL string for a legal file name
        radixCharCode (-1) -- an ASCII character code
        radixSwitch (-2) -- something that is either true or false
    num -- a signed integer
    str -- a BCPL string, e.g., let str = vec lString, "literal string"
    char -- an ASCII character code, i.e., 0 le char le #377
    sw -- a switch, i.e., sw eq true % sw eq false
    index -- a character position in a string
    dbl -- a double precision integer, e.g., let dbl = vec 1
    M1, P1 -- minus 1 and plus 1 respectively
```

II. Descriptions of Subroutines

// String manipulation

ValueToString (value, destinationStr, [radix/10]) -> destinationStr

Convert value to a string according to the radix and put that string in destinationStr.

StringToValue (sourceStr, [radix/10, [resultValue]]) -> value

Convert sourceStr to a value according to the radix. Put the value into resultValue if and only if radix specifies a multiword type thing.

CopyString (sourceStr, destinationStr) -> destinationStr

Copy sourceStr into destinationStr.

AppendChar (char, destinationStr) -> destinationStr

Append char onto destinationStr.

AppendString (sourceStr, destinationStr) -> destinationStr

Append sourceStr onto destinationStr.

AppendNum (value, destinationStr, [radix/10]) -> destinationStr

Convert value into a string according to radix and append it onto destinationStr.

MakeString (destinationStr, radix,value, [radix,value, ...])
-> destinationStr

Make up a string in destinationStr. Convert each of the values into a string according to its paired radix and concatenate the strings.

ImbedChar (char, destinationStr, [index/destinationStr>>SL+1])
-> destinationStr

Imbed (insert) char in destinationStr at the position specified by index.

ExtractString (sStr, dStr, beginIndexM1, [endIndexP1/sStr>>SL+1])
-> dStr

Make a string in dStr from the characters in sStr from beginIndexM1 to endIndexP1 exclusive.

SearchChar (searchStr, forChar, [beginIndexM1/0]) -> index/0

Search searchStr for forChar beginning at character position beginIndexM1 + 1. If found, return the index, otherwise return 0.

SearchString (searchStr, forStr, [beginIndexM1/0, [capSw/false]]
-> index/0

Search searchStr for forStr beginning at character position beginIndexM1 + 1. If found, return the index, otherwise return 0. If capSw, ignore capitalization.

StringEqual (str1, str2, [capSw/false]) -> true/false

Decide whether or not str1 eq str2. If capSw, ignore capitalization.

// Miscellaneous

Sc (num1, num2) -> -1/0/1

> You may not know it, but (exp relation exp) doesn't work correctly if the two expressions differ by more that 2**15. This routine works correctly for all values of num1 and num2. The results are the same as with Usc, i.e., -1 if num1 ls num2, 0 if num1 eq num2, and 1 if num1 gr num2.

Abs (num) -> num

> = (num ls 0 ? -num, num)

Max (num1, num2) -> num

> = (Sc (num1, num2) ge 0 ? num1, num2)

Min (num1, num2) -> num

> = (Sc (num1, num2) le 0 ? num1, num2)

MinMax (minNum, num, maxNum) -> num

> = Min (maxNum, Max (minNum, num))

InBounds (minNum, num, maxNum) -> true/false

> = Sc (minNum, num) le 0 & Sc (num, maxNum) le 0

IntDivide (dividend, divisor) -> num

> = (dividend + divisor - 1) / divisor

ZoneLeft (zone) -> available memory size

> Return the size of the largest buffer left in zone.

WriteForm (stream, radix,value, [radix,value, ...])

> Convert each value to a string according to its paired radix and write it to stream.

// Double precision arithmetic

DblMul (multiplicand1, multiplicand2, dblResult) -> dblResult!1

> dblResult ← multiplicand1 * multiplicand2

DblDiv (dblDividend, divisor, dblResult) -> dblResult!1

> dblResult ← dblDividend / divisor

DblAdd (dblAddend1, dblAddend2, dblResult) -> dblResult!1

> dblResult ← dblAddend1 + dblAddend2

DblSub (dblMinuend, dblSubtrahend, dblResult) -> dblResult!1

dblResult ← dblMinuend - dblSubtrahend

DblSingAdd (dblAddend, addend, dblResult) -> dblResult!1

dblResult ← dblAddend + addend

DblMulAdd (multiplicand1, multiplicand2, addend, dblResult)
-> dblResult!1

dblResult ← (multiplicand1 * multiplicand2) + addend

DblMulDiv (multiplicand1, multiplicand2, divisor, [dblResult])
-> dblResult!1

dbl ← (multiplicand1 * multiplicand2) / divisor; if numargs eq 4, dblResult ← dbl

III. List of Subroutines

```
// String manipulation

ValueToString (value, destinationStr, [radix/10]) -> destinationStr
StringToValue (sourceStr, [radix/10, [resultValue] ]) -> value
CopyString (sourceStr, destinationStr) -> destinationStr
AppendChar (char, destinationStr) -> destinationStr
AppendString (sourceStr, destinationStr) -> destinationStr
AppendNum (value, destinationStr, [radix/10]) -> destinationStr
MakeString (destinationStr, radix,value, [radix,value, ...])
    -> destinationStr
ImbedChar (char, destinationStr, [index/destinationStr>>SL+1])
    -> destinationStr
ExtractString (sStr, dStr, beginIndexM1, [endIndexP1/sStr>>SL+1])
    -> destinationStr
SearchChar (searchStr, forChar, [beginIndexM1/0]) -> index/0
SearchString (searchStr, forStr, [beginIndexM1/0, [capSw/false] ]
    -> index/0
StringEqual (str1, str2, [capSw/false]) -> true/false

// Miscellaneous

Sc (num1, num2) -> -1/0/1
Abs (num) -> num
Max (num1, num2) -> num
Min (num1, num2) -> num
MinMax (minNum, num, maxNum) -> num
InBounds (minNum, num, maxNum) -> true/false
IntDivide (dividend, divisor) -> num
ZoneLeft (zone) -> available memory size
WriteForm (stream, radix,value, [radix,value, ...])

// Double precision arithmetic

DblMul (multiplicand1, multiplicand2, dblResult) -> dblResult!1
DblDiv (dblDividend, divisor, dblResult) -> dblResult!1
DblAdd (dblAddend1, dblAddend2, dblResult) -> dblResult!1
DblSub (dblMinuend, dblSubtrahend, dblResult) -> dblResult!1
DblSingAdd (dblAddend, addend, dblResult) -> dblResult!1
DblMulAdd (multiplicand1, multiplicand2, addend, dblResult)
    -> dblResult!1
DblMulDiv (multiplicand1, multiplicand2, divisor, [dblResult])
    -> dblResult!1
```

VMEM, a virtual memory package for the Alto

***** Note: there has been a change in the division of VMEM procedures among the .BR files. See the last section of this writeup for details. *****

The VMEM package provides a virtual memory facility for Alto programs. The virtual address space is 2↑24 words; the page size is 2↑8 (256, 400b) words.

The package uses several data structures for which you (the user) must supply storage, as follows:
1) A hash map, whose size is 2P+1 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) An optional logging area, located just below the hash map. If desired, VMEM will make an entry in this area each time you make a reference to a virtual address, and call a procedure when the area fills up.
3) A buffer pointer table of 256 words.
4) 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 at an address which is a multiple of the page size (400b) and is a multiple of the page size long.
5) A locked cell list of 2N+2 words, where N is the largest number of cells you will ever want to use as locks (see below).

VMEM is designed to use special microcode loaded into the Alto microinstruction RAM, although it will run properly without such microcode. Unfortunately, there is no straightforward procedure for getting the relevant microcode into the RAM and getting it properly hooked up to the Nova emulator, if it is to share the RAM with any other special microcode. People wishing to use the RAM with VMEM should be prepared to include the microcode source in their own microprograms.

1. Initialization

VmemRam()
VmemSoft()

Before calling InitializeVmem, you must call one of these two procedures to tell VMEM whether or not you are using the RAM. After calling InitializeVmem, you may call either of these procedures at any time if you want.

InitializeVmem(HMAP, HMAPSIZE, BPTAB, LCL, LLCL, MSBASE, MSPROC[, NBPROC])

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. MSBASE is the base of the logging area (below HMAP), or 0 if no logging is desired. MSPROC is the procedure to call when the logging area fills up (see below). NBPROC is an optional procedure to call when VMEM cannot find enough unlocked buffers to handle a page fault or a SnarfBuffer call (see below): VMEM will call NBPROC and then try again, indefinitely. If NBPROC is not supplied, VMEM will call Swat instead.

AddBuffers(FIRST, LAST)

In order for VMEM to function, you must give it 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.

For implementation reasons, virtual pages -8 through -1 are not legal. If you try to read from page -1, you will get back unspecified data. If you try to read from pages -8 through -2, or write in any of these pages, VMEM will call Swat.

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

MSPROC(ARG, N[, VP]) [MSPROC from InitializeVmem]

If N<0, ARG is a core page number (i.e. a core address divided by 400b), and the type of event depends on N as follows:
N=-1: page ARG is being freed because it is needed for some other purpose than holding its current page of data. VP is the virtual address currently in the page.
N=-2: page ARG, formerly not available to VMEM, has now become available (through AddBuffers or UnsnarfBuffer).
N=-3: page ARG, formerly available to VMEM, has now become unavailable (through SnarfBuffer).

If N>=0, ARG is the MSBASE argument to InitializeVmem or InitSoftVmem, and N words (N/2 entries) starting at ARG contain 2-word entries representing calls on the address mapping functions. Each entry consists of a 24-bit virtual address with the top 8 bits unused: no distinction is currently made between reads and writes. If you are not using the RAM, VMEM will start reusing the area starting at MSBASE; however, if you are using the RAM, VMEM cannot determine the correct value of N (and will call MSPROC with N=0), so MSPROC must return this value and reset the R or S register itself.

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 in the hash map 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.

VirtualPage(CPAGE)

Returns the virtual page currently occupying core page CPAGE. Returns -2 if CPAGE is currently empty, or -3 if CPAGE is unavailable to VMEM. If CPAGE is not in the range 0 to 377b inclusive, returns garbage.

SnarfBuffer(BUFPTR[, NBUFS, ALIGN])

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, or 0 meaning any buffer(s) will do and SnarfBuffer should find it (them). The effect of SnarfBuffer is to remove NBUFS (default is 1) buffers starting with that buffer from use by VMEM. A typical application of SnarfBuffer is to acquire space for display data or Ethernet buffers.

If BUFPTR is non-zero and some buffer in the specified range is locked (see below), SnarfBuffer returns 0; normally SnarfBuffer returns the address of the buffer.

If you need a group of buffers aligned as described under PageGroupAlign below, you may also supply an ALIGN argument, which works the same way as the value returned by PageGroupAlign.

UnsnarfBuffer(BUFPTR)

Reverses the action of SnarfBuffer. If you acquired a range of buffers, you must return them one at a time with UnsnarfBuffer.

LockCell(LVLOCK, PROC)

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.

When the virtual memory system wants to change the contents of a buffer, it goes through the lock list and calls PROC(LVLOCK, NEWADDR, false) for each lock cell which contains a pointer into the buffer, where NEWADDR is the proposed new core address for the page (if it is just being moved around in core, e.g. to make room for a page group) or 0 (if it is being written out). If any PROC returns false, the system will refrain from the proposed action. If all PROCs return true, the system calls PROC(LVLOCK, NEWADDR, true) for each appropriate lock cell, and updates the contents of the lock cell (zeroing it if the page is being written out) in the process. Note that in the latter case, the lock cell will NOT be restored automatically if the page is read back in at some future time.

The number of different lock cells is limited to the parameter LLCL supplied to InitializeVmem, divided by 2, minus 1. If the lock list is full, LockCell calls Swat.

The system provides the procedures LockOnly, LockReloc, and LockZero, described below, simply because they are useful default actions: the user may provide an arbitrary procedure for PROC.

LockOnly(LVLOCK, NEWADDR, FLAG)

If the PROC parameter of LockCell is LockOnly, the system will not move or write the page.

LockReloc(LVLOCK, NEWADDR, FLAG)

If the PROC parameter of LockCell is LockReloc, the system may move the page in core (updating the lock cells), but will not write it out.

LockZero(LVLOCK, NEWADDR, FLAG)

If the PROC parameter of LockCell is LockZero, the system may move or write the page whenever necessary, zeroing the lock cell in the latter case.

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, FLAG)

If PTR is a pointer into a locked buffer, returns true, otherwise returns false. If FLAG=true, IsLocked returns true even if there are locked pointers into the same buffer as PTR, provided that the relocation procedures are willing to have the buffer swapped out; if FLAG=false or FLAG is absent, IsLocked only returns true if there are no locked pointers to the buffer whatever.

Note that if the page addressed by PTR itself is not locked, IsLocked will return false even if there exist locked pointers to other pages in a page group which PTR points into.

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.)

5. User routines

The VMEM package does not assume any particular correspondence between virtual addresses and disk pages, or indeed that you are using the disk at all: for example, you can use the Ethernet for paging if this suits your fancy, or store the data in some compressed form on the disk. Consequently, you must supply a number of routines to establish the correspondence between virtual page addresses and stored data.

CleanupLocks()

This routine is called on every page fault, and at other times when VMEM needs to know that the contents of the lock cells are correct. Normally, CleanupLocks need not do anything; however, if you have pointers in microcode registers or other non-standard places which point into page buffers, CleanupLocks should copy them into lock cells known to VMEM.

PageType(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. PageType must return 1 if the page is an existing page, or -1 if a new page. If VPAGE is invalid, PageType can do whatever it wants, but it should not return.

PageGroupBase(VPAGE)
PageGroupSize(VPAGE)

These routines are for applications where it is necessary to cause a group of pages, rather than a single page, to always be transferred into and possibly out of core at the same time and to occupy consecutive page buffers. PageGroupBase must return the virtual page number of the first page in the group; PageGroupSize must return the size of the group. If you are not using page groups, PageGroupBase should return its argument, and PageGroupSize should return 1.

VMEM distinguishes between read groups, in which individual pages may be rewritten if they become dirty, and write groups, in which the entire group must be rewritten if any page becomes dirty. For write groups, PageGroupSize must return the negative of the size of the group.

PageGroupAlign(VPAGE)

Occasionally it is necessary to align a page or group of pages so that some of the bits of the core address are zero; for example, if you want to get the effect of 1000b-word pages, it is necessary to align each group so that the 400b-bit of its core address is zero. PageGroupAlign should return a mask which specifies which of the high-order 8 bits of the core address must be zero; in the example, PageGroupAlign should return 1. For pages which do not require alignment (the usual case), PageGroupAlign should return 0.

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.

6. Standard use

The standard use of VMEM is to do swapping on a standard disk file in which virtual page N corresponds to file page N+2 (page 1 is reserved for use as an index, and page 0 is the leader page), using the ISF package (described elsewhere) to obtain rapid random access to the file. The following program fragment will accomplish this, assuming you are just using 400b-word pages in the most straightforward way.

```
external          // entries for VMEM
[       CleanupLocks
        PageType
        PageGroupSize
        PageGroupBase
        PageGroupAlign
        DOPAGEIO
]

external          // links to ISF
[       InitFmap
        IndexedPageIO
]

static
[       MyFmap  // pointer to work area for ISF
]


// To initialize ISF, set MyFmap to point to a work area
// of size MyFmapLength, and then call
//      InitFmap(MyFmap, MyFmapLength, FilePtr, true)
// where FilePtr is a FP (see the O.S. manual)
// for the paging file. A reasonable value for
// MyFmapLength is 80 -- see the ISF writeup.


let CleanupLocks() be [ ]

let PageType(vp) = 1

let PageGroupSize(vp) = 1
let PageGroupBase(vp) = vp
let PageGroupAlign(vp) = 0

let DOPAGEIO(vp, core, np, wflag) be
[
        IndexedPageIO(MyFmap, vp+2, core, np, (wflag? -1, 1))
]
```

7. Packaging

The VMEM package actually consists of several files:
VMEM.BR - the code required to process page faults, plus LockCell and UnlockCell
VMEMAUX.BR - all the other entries to VMEM, except InitializeVmem
VMEMINIT.BR - InitializeVmem
VMEMA.BR - a small amount of assembly-language code
VMEMSOFT.BR - a software version of the VMEM microcode
VMEM.USE - the program fragment listed above
VMEM.MU - the VMEM microcode.
You must load VMEM, VMEMAUX, VMEMINIT, and VMEMA with your program, and also VMEMSOFT if (as is normally necessary) you are not using the RAM. In addition, you must load the ISF package (files ISF.BR and ISFINIT.BR) if you are using VMEM in the standard manner described above. Once you have called InitializeVmem, you may throw away VMEMINIT; once you have done all your calls on AddBuffers, etc., you may throw away VMEMAUX.