A UNIX PORT OF THE PERKIN-ELMER PASCAL RUN-TIME LIBRARY

by

HARVARD CHARLES TOWNSEND

B.S., Kansas State University, 1980

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

Approved by:

[illegible]

Major Professor

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

This paper was submitted as partial fulfillment for the degree of Master of Science in Computer Science at Kansas State University, Manhattan, Kansas. I first of all wish to thank my major professor, Dr. Virgil E. Wallentine, for excessive patience and confidence in my ability to complete this project. All programmers should be so lucky as to have a supervisor like him. Secondly, I could not have even begun to tackle this project without the guidance and expert technical consultation of Robert Young. All "real" programmer apprentices should be so lucky as to have a mentor such as he. Finally, my deepest gratitude is due my wife, Dana, and my children, Sarah and Michael, who put up with many long hours of me staring at the monitor and pecking at the keyboard. All husbands and fathers should be so fortunate as to have a wife and family as precious as mine.

1. INTRODUCTION

1.1. Motivation and Language Description

As the UNIX operating system [1] has increased in popularity, the desire to port a variety of languages to the UNIX environment has also increased. The programming language Pascal [2,3], although still largely academic, likewise has enjoyed increased use. Although a number of implementations of Pascal on UNIX exist [4,5,6], the Perkin-Elmer Corporation (PE) at the time of this project did not have a Pascal compiler available to market with the version of UNIX they sell for their minicomputers. Neither did the Kansas State University (KSU) Department of Computer Science have a compiled standard Pascal for their Perkin-Elmer minicomputers running Edition 7 UNIX. PE did, however, have a standard Pascal implementation [7] available for their own OS/32 multitasking operating system [8]. Consequently, PE requested that the KSU Dept. of Computer Science port their OS/32 Pascal onto the version of UNIX which runs on their minicomputers.

Perkin-Elmer Pascal (PEPascal) is marketed for PE's 32-bit processor family running OS/32 R05.2 or higher. It is an implementation of the standard Pascal language with a number of useful extensions. PE provides an optimizing 10-pass compiler in both overlay and resident task versions. The compiler driver and passes are written in PEPascal. The compiler provides a number of programming aids in the form of options to the compiler. It is possible to get a listing of the compiled program, a cross reference of the program identifiers, a summary listing, assembly listing, and an object map with these options. The user can also process a number of source programs at once with the

BATCH option. Although these options add to the size and complexity of the compiler passes, they do help make PEPascal a good implementation language for development applications.

They also provide a library of run-time routines (RTL) to support the PEPascal task while it is executing. These routines are written in PE's Common Assembly Language (CAL) [9]. The RTL routines perform a wide variety of functions, including memory allocation, error handling, and implementation of standard procedures and functions. It also has the code which implements a number of extensions to the standard language which provide the user with easy access to OS/32 services and utilities. To the user, these extensions are provided in the form of either a "prefix" to the program source code or external procedures declared with the EXTERN directive.

1.2. Language Portability

A large proportion of the literature on software portability deals with porting across machine architectures [10,11]. Porting to a new machine normally means porting to a new operating system (OS), so many of the same strategies apply. Furthermore, nearly all language implementations work on top of a host OS, so the underlying OS may have more of an influence on the porting process than that credited to it in the literature.

A number of approaches have been proposed for both writing portable software or just moving software to a different environment [10,11]. One method is to use tools which are available in a variety of environments. A good example of such a tool is a compiler or interpreter for a popular language. If you write your software in a high-level language such as C, Pascal, FORTRAN, or COBOL, you are likely to find a translator for that language available on the target system. Then the porting process is simply to recompile/reinterpret the source on the new machine. If the user is careful to only use the language's standard facilities, then little, if any, changes will be needed. Since PEPascal's compiler driver and passes are written in an extended Pascal standard, it does employ this portability strategy in part. However, several of the language extensions used by the compiler are very specific to OS/32 and the RTL is still written in host-system dependent assembly language.

Another approach to portable software is to implement it with a flexible method which can be performed by a large number of tools. This way, an implementor is likely to be able to find some suitable tool in the new environment which may be similar or even identical to the tool with which the original software was developed. This is the notion of macro processors. This, for example, is how SNOBOL4 [12] was implemented. Its syntax is sufficiently simple that most any macro processor can translate it. This method does not apply to PEPascal, however.

A third approach to portability is the "abstract machine" approach. With this, a fixed host-independent language is translated into some target language (abstract code). The target language is designed for an abstract machine which is ideally suited for the solution to this problem. To implement the software on a real machine, the target language must be interpreted into something understood by the new host. This method is employed by a number of high-level language systems, such as Per Brinch Hansen's Concurrent Pascal [13], KSU's implementation of Concurrent Pascal [14], and the Portable Simula system [15]. The abstract code is often translated into assembly code of the host machine, but in some cases it is translated into another high-level language supported by the new host. Whitesmiths' Ltd., for example, uses this strategy to port Pascal onto UNIX. Their implementation translates the Pascal source into the C programming language [16]. Since C is the standard language for UNIX, most any UNIX system would be able to compile the abstract code into code suitable for the host machine. The only thing which would have to be ported would be the Pascal-to-C translator. If this is written in C, then moving this system to a new environment would be trivial. Whitesmiths' Ltd. claims that this method is beneficial since the code is now run through an optimizing C compiler. However, this method has the overhead of a second lexical analysis.

The abstract machine approach is applicable to the PEPascal implementation. In the PEPascal compiler, the first five passes perform the lexical and semantic analysis. If no errors occur, abstract code is produced from pass5 which can then be translated into the host-dependent code. Passes 6 through 9 of the compiler do this by translating this intermediate code into Perkin-Elmer machine code. Moving the system to a new machine would thus require changing passes 6 - 9 to produce the host machine code, unless a Whitesmiths-like technique was used. Since our application did not port the language onto a different machine, there was clearly no need to change PE's approach to producing machine code.

All these methods have in common the fact that they are attempting to reduce the amount of system-dependent features which must be changed when moving to a new environment. PEPascal further aids this process by isolating the system-dependent features of the implementation in RTL routines written in CAL. Once the RTL was working on UNIX, then the port would be nearly complete since the compiler driver and passes are all written in PEPascal. This would not be possible without the UNIX utility `cvobj' which translates OS/32 object code into UNIX object code. With this utility, we were able to produce UNIX versions of the object code for the driver and passes from the OS/32 objects supplied by PE. Once the RTL was ported, then, we would have a working compiler. Only then could we make changes to the compiler source and recompile it. Consequently, the first phase of the PE porting project was to port the RTL as described in this report.

2. PORTING CONCEPTS

2.1. Interface With the Underlying OS

By design, the routines in the PEPascal RTL are not directly accessible from a high-level user program. Instead, the compiler generates external references to these RTL routines which are resolved during link-editing. Each RTL routine then has a specific function to perform. To accomplish this, many of them require calling other RTL routines and/or services provided by the underlying OS. In this way, the RTL acts as the interface between the user program and the host OS. This relationship is depicted in Figure 1 for a generic multitasking OS.

The OS services requested by the RTL routines thus represent one of the system-dependent features of the language environment which must be ported. I will therefore compare how a user process requests OS services under OS/32 and UNIX as a prelude to the actual implementation details described in section 3 below.

2.1.1. OS/32 Interface

In OS/32, OS services are requested with supervisor calls (SVC) [8, chapter 5]. These SVCs are classified by a decimal number between 0 and 15 which specifies the type of call. SVC 1, for example, handles file I/O requests, SVC 2 does a number of general purpose functions, and the SVC 7 group performs file management services. SVC 0 is reserved for user-made system extensions. Table 1 shows the complete list of SVC services available in OS/32.

These SVCs are actually made with an "svc" instruction from an

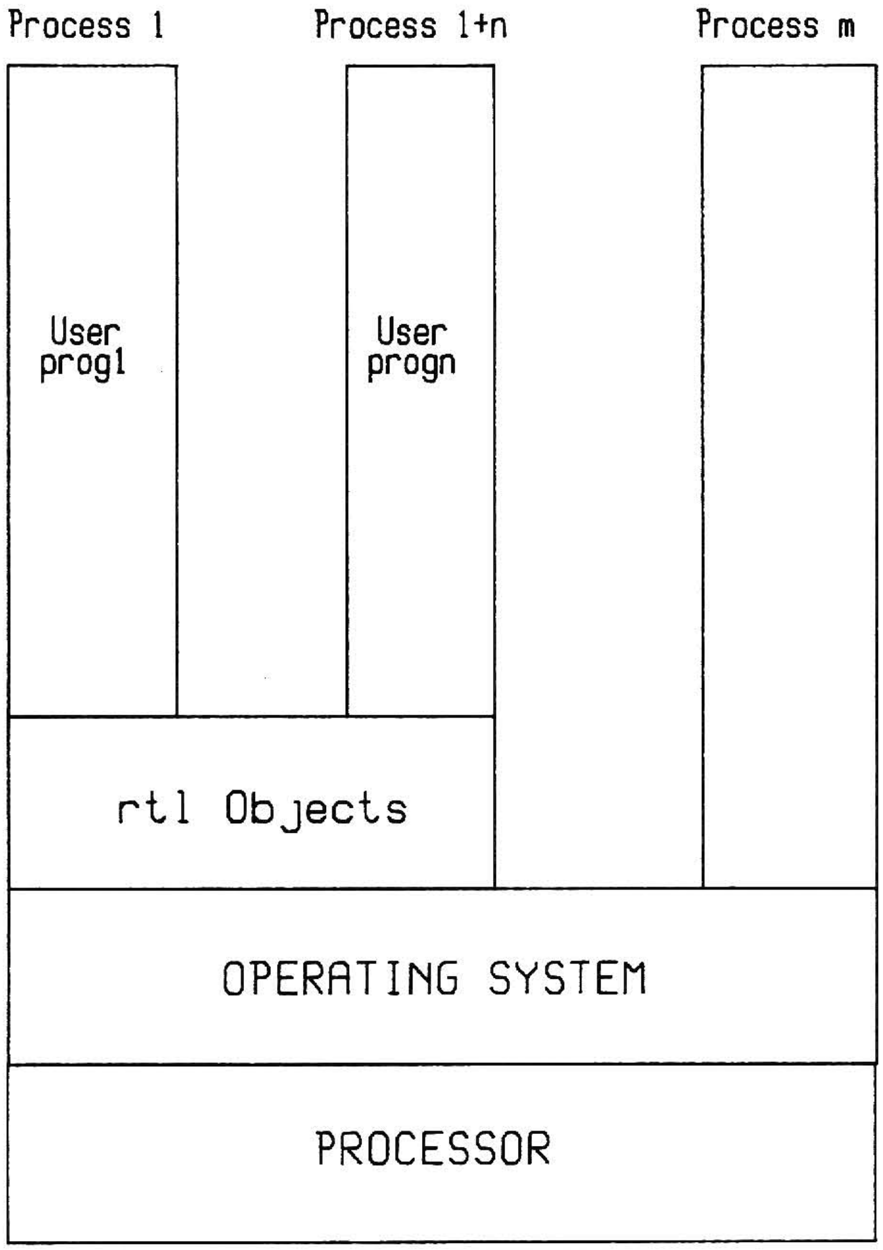


Figure 1. Implementation of Perkin-Elmer Pascal in a multitasking operating system.

n = number of active PE Pascal processes
m = total number of active processes in system.

Table 1. OS/32 Supervisor Calls (from [6]).

Call type	Function
SVC 0	Reserved for user-made system extensions
SVC 1	Input/output request
SVC 2 Code 1	Pause the task
SVC 2 Code 2	Get storage for task's impure segment
SVC 2 Code 3	Release storage reserved with SVC 2 Code 2
SVC 2 Code 4	Set status in PSW
SVC 2 Code 5	Fetch pointers - update the UDL
SVC 2 Code 6	Convert binary number to ASCII hex or decimal
SVC 2 Code 7	Log message
SVC 2 Code 8	Fetch current time-of-day into a buffer
SVC 2 Code 9	Fetch date into a buffer
SVC 2 Code 10	Time-of-day wait
SVC 2 Code 11	Time interval wait
SVC 2 Code 15	Convert ASCII hex or decimal to binary
SVC 2 Code 16	Pack file descriptor
SVC 2 Code 17	Scan mnemonic table
SVC 2 Code 18	Move ASCII characters in memory
SVC 2 Code 19	Peek at user-related task/system information
SVC 2 Code 20	Expand allocation
SVC 2 Code 21	Contract allocation
SVC 2 Code 23	Timer management facilities
SVC 2 Code 24	Set accounting information
SVC 2 Code 25	Fetch accounting information
SVC 2 Code 26	Fetch device mnemonic
SVC 3	End-of-task
SVC 5	Fetch overlay
SVC 6	Intertask coordination
SVC 7	File handling services
SVC 9	Load Task Status Word (TSW)
SVC 14	Reserved as a user SVC
SVC 15	ITAM device dependent I/O

assembly program. The first operand is the SVC type number. The second is the address of the corresponding SVC parameter block used to communicate values between the calling program and the OS. The parameter block has a specific length and format based on the type of service requested. An SVC 1 parameter block, for example, has fields for the starting and ending address of the buffer used for transferring data in the I/O request. Figure 2 shows this structure.

Since the SVC is a machine-level instruction, a user clearly cannot make such a call directly from a Pascal program (see Figure 3a). This assures that the system-dependent features of the language are protected in the assembly-level RTL. However, it has been stated that any reasonable implementation of the Pascal language must allow the programmer access to system calls and other operating system utilities [6]. Without this, the language would not be suitable for most useful applications. With this in mind, Perkin-Elmer Pascal allows the user to make SVC calls from his/her Pascal program. The language provides a complete set of external procedures for this purpose which must be declared with the EXTERN directive as an extension to the standard language [3]. Again it is important to note that RTL routines perform the actual calls to the OS/32 kernel. These external routines are merely a convenient user interface. A language could also provide this facility if it allowed linking to user-written assembly language routines.

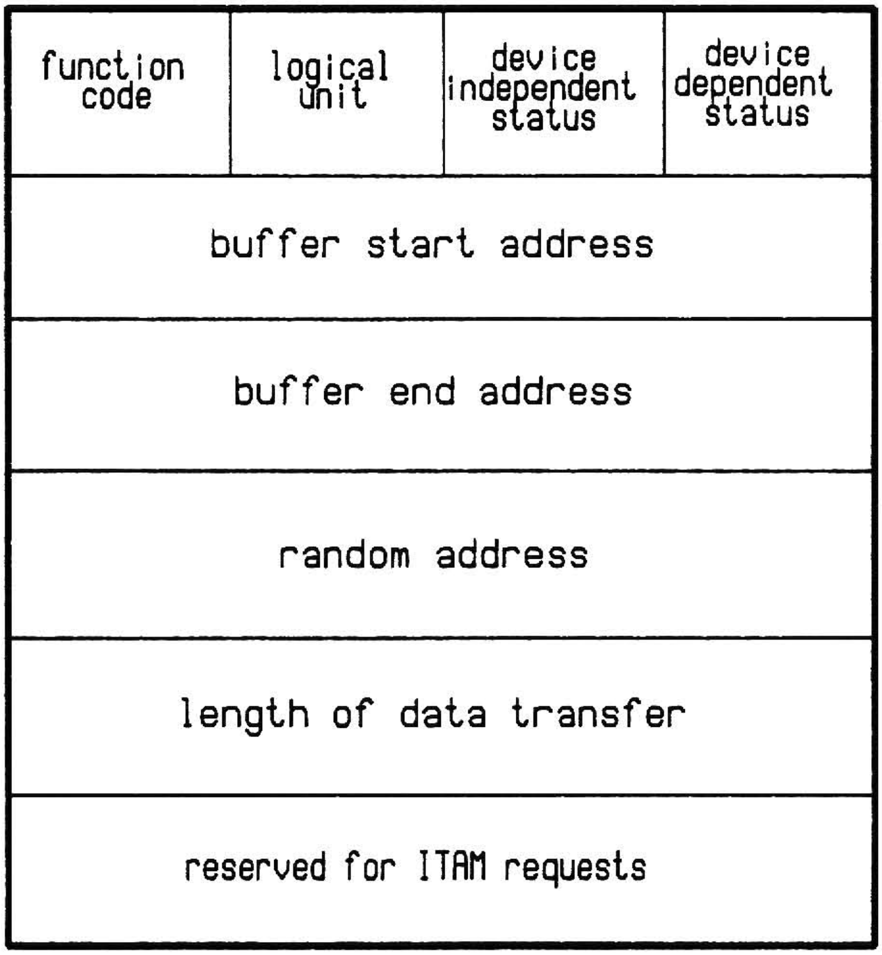
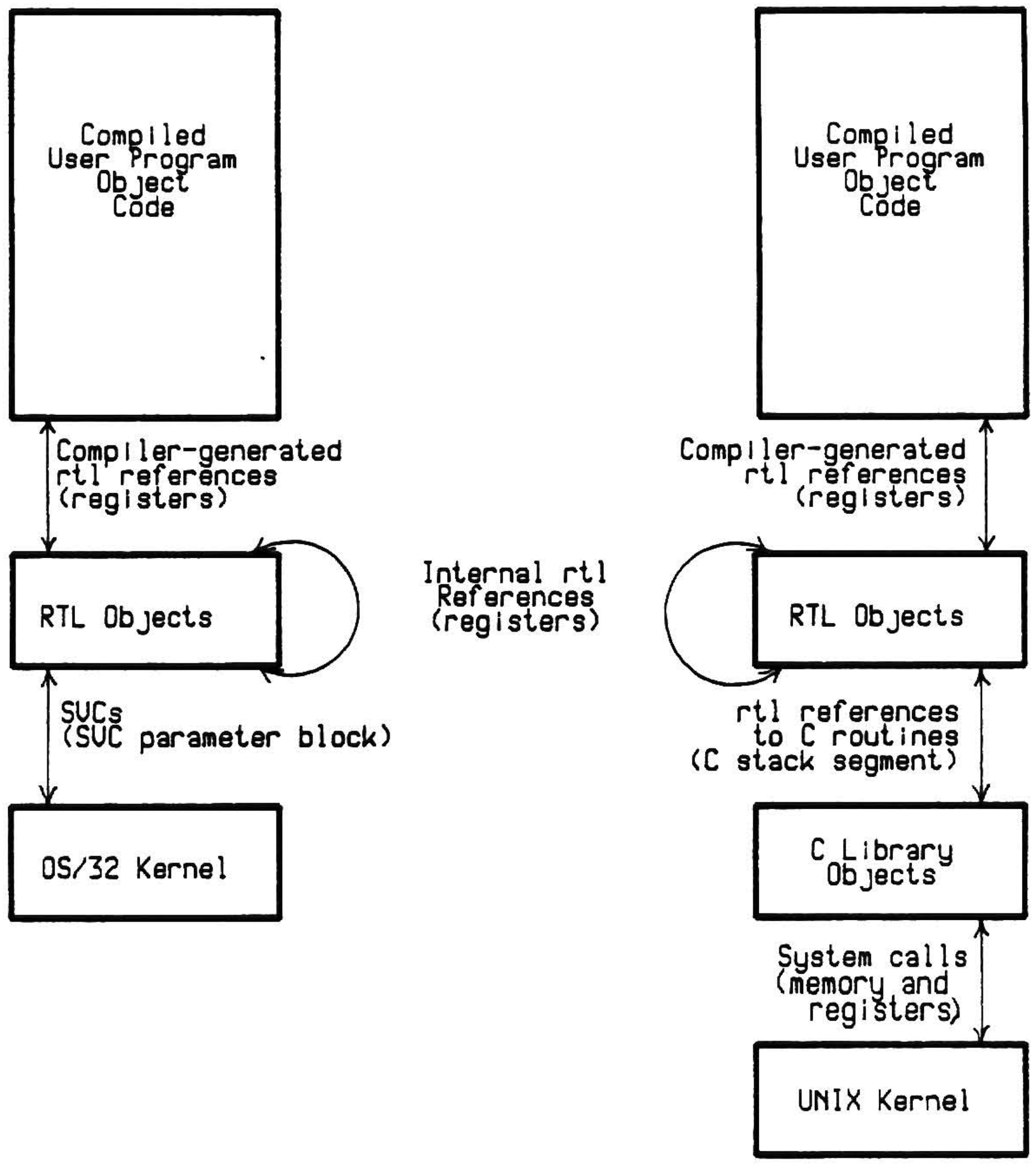


Figure 2. SVC 1 parameter block.


Compiled User Program Object Code
Compiler-generated rtl references (registers)
RTL Objects
Internal rtl References (registers)
SVCs (SVC parameter block)
OS/32 Kernel
Compiled User Program Object Code
Compiler-generated rtl references (registers)
RTL Objects
rtl references to C routines (C stack segment)
C Library Objects
System calls (memory and registers)
UNIX Kernel

a. OS/32 environment. b. UNIX environment.

Figure 3. The relationship between a compiled Perkin-Elmer Pascal program and the underlying OS at run-time. Arrows represent the nature of calls to uncerlying layers. The structure(s) for communicating data are shown in parentheses.

2.1.2. UNIX Interface

Like OS/32, communication with the UNIX supervisor occurs via SVC calls. I will refer to these as "system calls" rather than SVCs for two reasons: 1) help distinguish between OS/32 and UNIX service requests, and 2) the UNIX documentation refers to them as system calls [17]. The services provided by UNIX system calls are quite different from the OS/32 services. Table 2 lists the services available in Edition 7 UNIX. A complete description of these calls and their interfaces is in Section 2 of Vol. 1 of the UNIX programming manuals [17].

Again like OS/32, the actual request for the OS service is made by an "svc" assembly instruction. The format of the operands is slightly different, however. Instead of an SVC type number, the first operand is always zero (0). The second operand is then the number which distinguishes the type of OS service requested. Even though the second argument is not a parameter block address as in OS/32, data is still communicated through a structure. Arguments to the system call immediately follow the "svc" instruction in memory (reserved and initialized with the CAL "define constant" instruction [9, page 3-30]). General register 0 may also be used for input. For example, the "write" system call (see Table 2) expects the file descriptor in register 0, the address of the buffer in memory immediately after the SVC instruction, followed by the number of bytes to be transferred. General registers 0 and 1 are then used to return values. Some require call-by-reference arguments for communicating values back to the calling routine.

As assembly language instructions, these system calls are not directly accessible from high-level language programs (again much like

Table 2. UNIX Edition 7 system calls.

access	- determine accessibility of file
acct	- turn accounting on or off
alarm	- schedule signal after specified time
break	- change core allocation
chdir	- change default directory
chmod	- change mode of file
chown	- change owner and group of a file
chroot	- change root directory
close	- close a file
creat	- create a new file
dup	- duplicate an open file descriptor
exec	- execute a file
exit	- terminate process
fork	- spawn a new process
fstat	- get file status
ftime	- get date and time
getgid	- get group identity
getpid	- get process identification
getuid	- get user identity
gtty	- control terminal device
indir	- indirect system call
ioctl	- control character special files
kill	- send signal to a process
link	- link to a file
lock	- lock a process in primary memory
lseek	- move read/write pointer
mknod	- make a directory or a special file
mount	- mount a file system
mpx	- create and manipulate multiplexed files
nice	- set program priority
open	- open a file for reading or writing
pause	- stop until signal
phys	- allow a process to access physical addresses
pipe	- create an interprocess channel
profil	- execution time profile
ptrace	- process trace
read	- read from a file
setgid	- set group ID
setuid	- set user ID
signal	- catch or ignore signals
stat	- get file status
stime	- set time
stty	- control terminal devices
sync	- update super-block
time	- get date and time
times	- get process times
umask	- set file creation mode mask
umount	- unmount a file system
unlink	- remove directory entry
utime	- set file times
wait	- wait for a process to terminate
write	- write on a file

OS/32). Conveniently, UNIX provides C language interfaces for all the system calls in addition to the assembly language interfaces [17,18]. The C routines which interface to the operating system are all part of the standard C library. A user program wishing to make a system call may thus simply call the corresponding C routine. This relationship is shown in Figure 3b. Arguments are passed in the process' stack segment in a specific order rather than via registers and specific memory locations. The "write" C routine, for example, expects the file descriptor on top of the stack, followed by the address of the buffer and the number of bytes to be transferred. The library routine then arranges the parameters appropriately for the actual system call. They may also perform a number of housekeeping functions such as setting the system error number if an error is detected. The "signal" system call in particular requires a large amount of processing before the actual OS call is made. It is thus most efficient in terms of programmer time to avoid the need to code this overhead by interfacing the UNIX operating system though calls to C library routines.

2.1.3 Relevance to the port

I have noted several similarities between the OS/32 and UNIX interfaces: they both occur via "svc" assembly instructions and are thus not directly accessible from high-level languages. Furthermore, although the format and size of data communication structures differ, parameters are passed at specific memory locations (UNIX also employs general registers 0 and 1). Two major differences influence the effort to port PEPascal from OS/32 to UNIX, however.

2.1.3.1 SVC translation

For one, the services offered by the two operating systems differ significantly. Although they both provide the fundamental services any operating system must provide - I/O control, file management, memory management, and process management - the manner in which they provide the services differs. There are thus a number of cases where no direct translation from OS/32 SVCs to UNIX system calls is possible. No UNIX system call exists to convert a number to an ASCII string, for example (the OS/32 SVC 2 code 6). Packing and unpacking a file descriptor (OS/32 SVC 2 code 16) is likewise foreign to UNIX. Several options are therefore available for porting. If a direct equivalent exists in UNIX, then the substitution is fairly straightforward. A UNIX "exit" call will terminate a process much like an OS/32 SVC 3. If no direct equivalent exists, the SVC can simply be emulated with CAL code with or without calls to appropriate C library routines. Finally, some SVCs have no application whatsoever in UNIX and therefore will require no translation. Packing and unpacking file descriptors is one service which will not need to be emulated in UNIX.

2.1.3.2 C stack vs. Pascal stack

The second difference between the two OS interfaces which affects the port is that in UNIX, OS calls are best accomplished via calls to C library routines. This forces the user program environment into relying on general register 7 as the pointer into the process' stack segment (C stack) since C routines expect this register to hold the address of its formal parameters. This has particular consequence in a Pascal programming environment because it must maintain 2 stacks: the Pascal stack used for local and global Pascal routine variables as well as the C stack for passing parameters to C routines (Figure 4).

In essence, the Pascal program will have to run in two different modes, the Pascal mode and the C mode. To complicate matters, they have completely different register conventions, the stacks grow in opposite directions (C stack down, Pascal stack up), and they work in different segments of the process' memory. The Pascal mode uses general registers 0, 1, and 2 as the stack pointers while the C mode uses register 7. Furthermore, UNIX only protects registers 8-15 across a routine call, implicitly protects register 7 since it is the stack pointer, and commonly uses register 0 for returning values. Pascal, on the other hand, only protects registers 0-2. Finally, the Pascal stack exists in the process' data segment while the C stack is in its stack segment. All this will therefore have to be taken into account when an RTL routine prepares to call a C routine. The details of how this was implemented in this port will be discussed in sections 3.2 to 3.5.

2.2. Memory Management

Another area of concern in porting the PEPascal language from OS/32 to UNIX is memory management. Since a process must work within the context of any number of other processes which reside coincidentally in memory (Figure 1), the manager of that memory will put constraints on the process' use of the memory allocated to it. It will thus be useful to examine the task memory management strategies of the two operating systems in order to understand the implementation needs of the port.

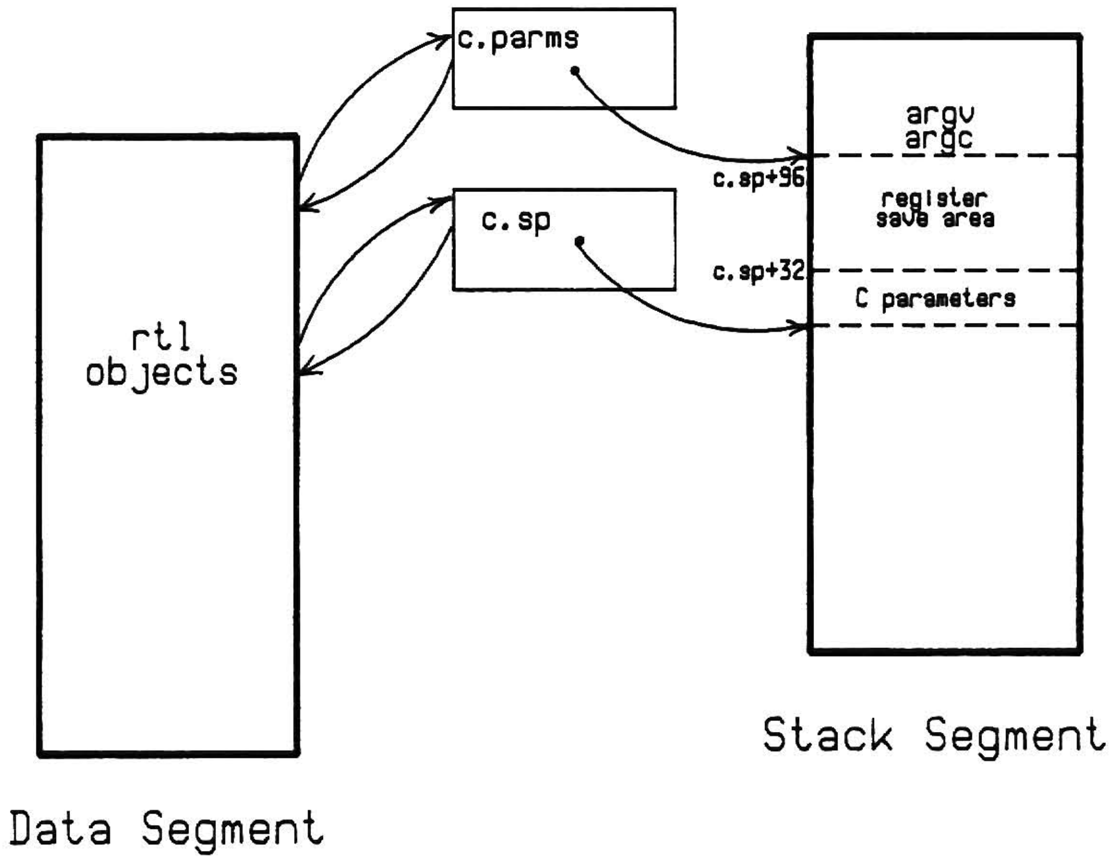


Figure 4. Management of the stack segment by a PEPascal process. Rtl routines fetch c.sp each time they switch to C mode. They then save registers and put parameters on the C stack in the areas indicated.

2.2.1. OS/32 task memory scheme

The OS/32 memory manager allocates memory to a user task when it is loaded. Memory is allocated on a first-fit basis from an area known as the "dynamic task memory space" [8]. This memory is in essence anything other than space reserved for system functions or other tasks. The task's memory is then deallocated when it reaches end-of-task.

The user task's memory can have up to 16 segments. A segment is defined as a set of contiguous addresses starting on a 64K boundary [8]. The segments are of 4 different types: impure, pure, task common, and reentrant library segments. The first 256 bytes of every task are protected as the User Dedicated Locations (UDL) [7,8]. At run-time, this area contains data used primarily for communication between the OS and the running task. This includes pointers to task space boundaries, data relative to OS-detected faults, and locations where old and new Task Status Words (TSW) are swapped in response to certain events.

The UDL occupies the beginning of the task's non-sharable impure segment. All tasks must have an impure segment. Three pointers, UBOT, UTOP, and CTOP, are associated with this segment (Figure 5) and stored in the UDL. UBOT is the starting address and is always relative address #0. UTOP is the address of the first fullword past the area reserved by the user task. Some tasks need an additional area in the impure segment, called the undefined area, for dynamic allocation. The address of the top of this area is available in CTOP, which is the top of the task's impure segment.

A task may also contain a write-protected pure segment which can be shared by other tasks. The "PURE" option in CAL produces the code for

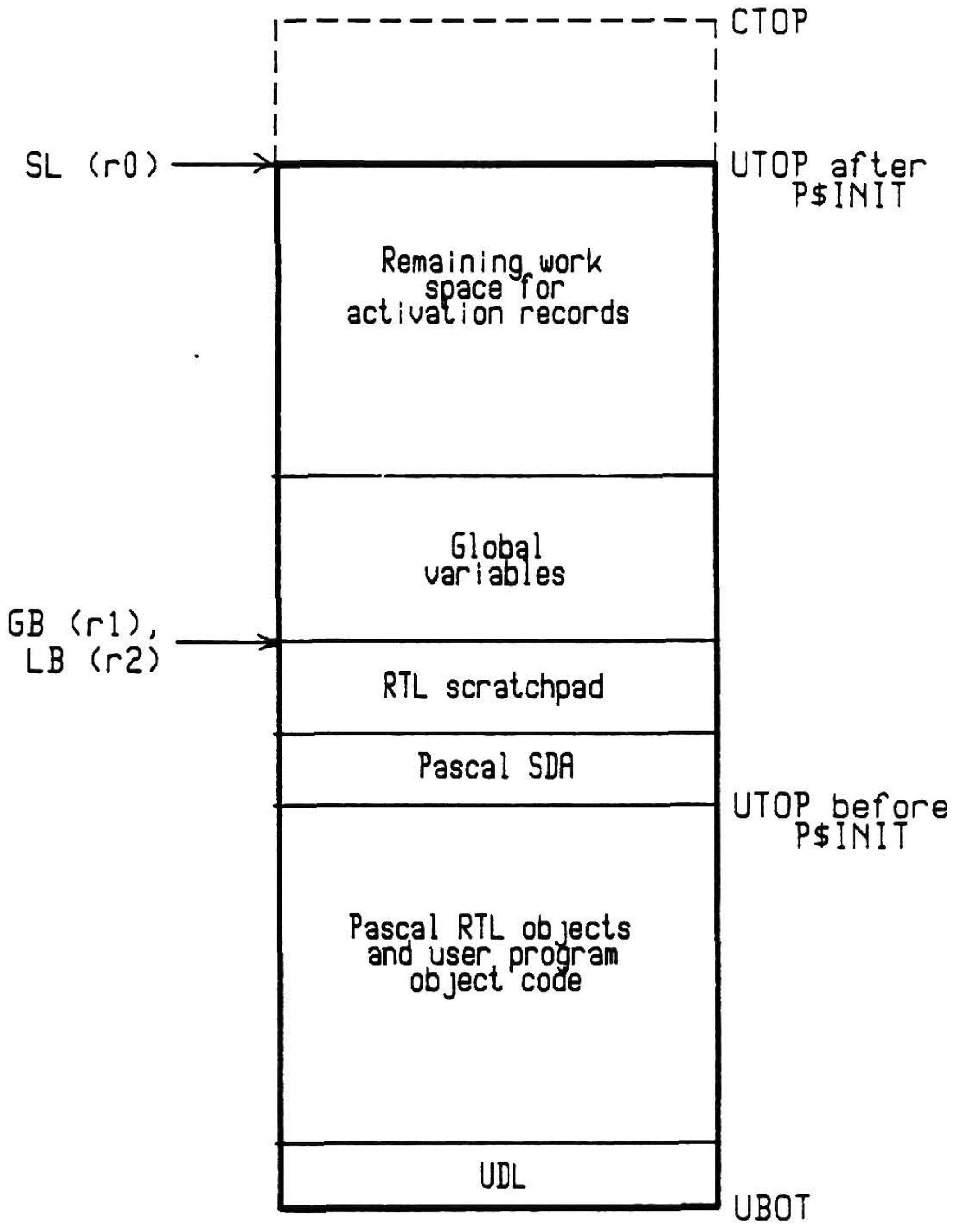


Figure 5. Initial Memory Map for Pascal program in an OS/32 environment.

SL = stack limit.
LB = local base.
GB = global base.

this segment. The shared reentrant library segments likewise are write-protected. The source of the code for these segments is some common object library rather than the result of some CAL option. Finally, tasks may share data areas with the task common segments. You can even set up protection on these common segments such that only one task can write to it while all others have read-only access.

2.2.2. UNIX process memory scheme

Like OS/32, a UNIX process' memory is allocated when the process is created. In UNIX, they are created by the "fork" system call [19]. Also, memory is allocated by the simple first-fit algorithm. The process gets the first free block of memory into which it will fit. A UNIX process has 3 major segments associated with it: the text, data, and stack segments (Figure 6). The process executes from the text segment which may contain shareable read-only code. If the process is related to the C library in any way (as a ported PEPascal process is), then the run-time initialization code, /lib/crt0.o, occupies logical address #0 of the process memory. This code serves to rearrange the arguments on the C stack (in the process' stack segment), branch to the label "main," and terminate the process if control ever returns from "main." The text segment also contains the executable user program object code and library objects. The data segment is private, holding both uninitialized and initialized data. This segment can be increased dynamically with the "sbrk" system call or the "malloc" C routine which calls "sbrk". The newly-allocated memory is initialized to zero. Finally, the stack segment starts at logical address #E1000 and grows down to the limit of #E0000. Its default size is thus 4K which can be

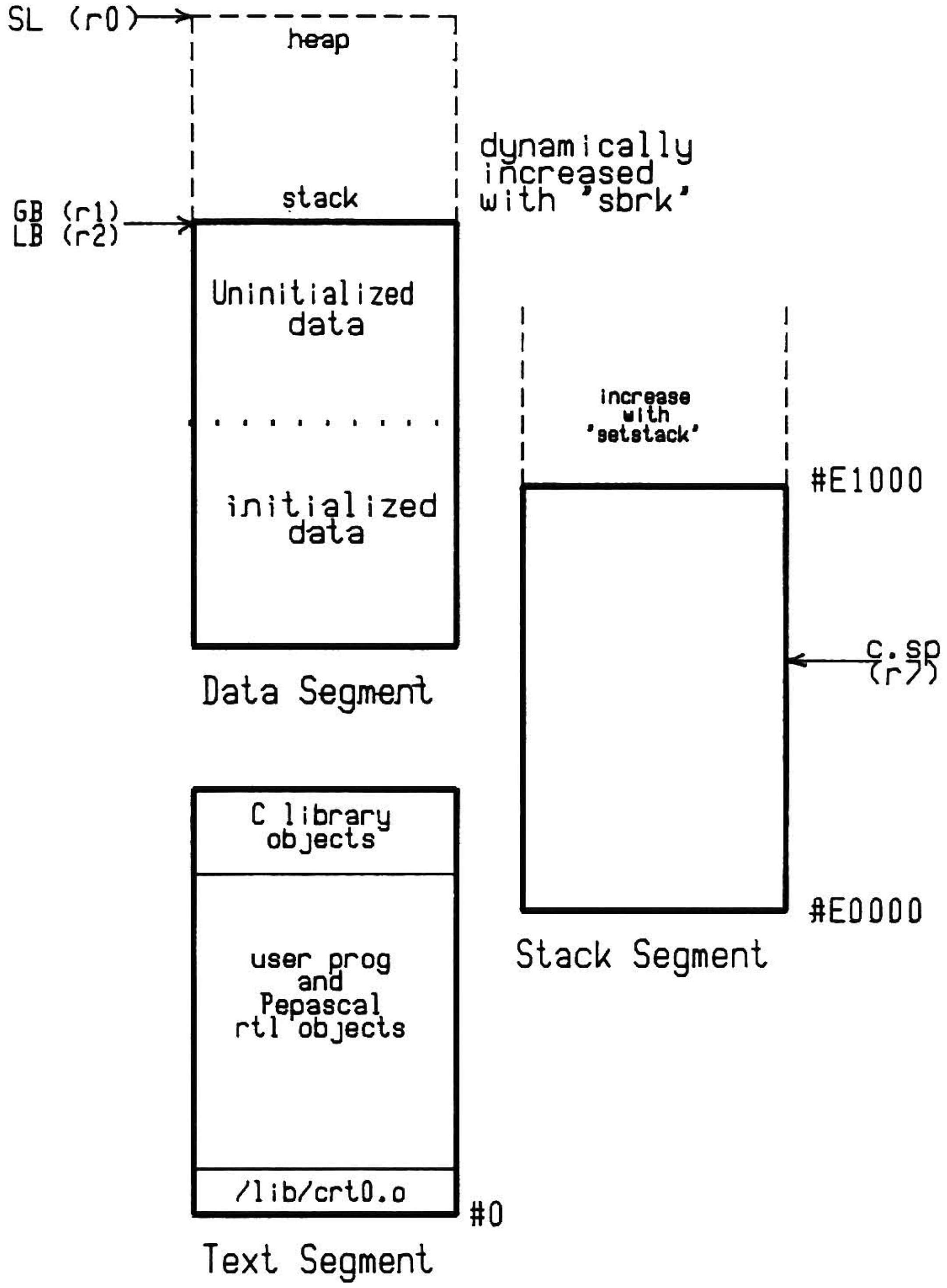


Figure 6. Memory map of a running PEPascal process in a UNIX environment. #0 is the relative starting address for the process.

altered by the `setstack' UNIX utility or the "-k" option of the UNIX link editor, `ld'.

2.2.3. Relevance to the port

At run-time, a PEPascal program needs some other areas in addition to the user program and library object code. First, it needs a Static Data Area (SDA) for special run-time variables. If FORTRAN routines are called from the PEPascal program, a variable number of fullwords must be allocated for FORTRAN Static Communications Area (SCA). An RTL scratchpad is also needed for local storage. Finally, it needs memory for global variables, dynamically allocated variables (the heap) and local variables for nested/recursive subroutines (the stack). This is what constitutes the Pascal stack described above in section 2.1.3.2.

Since the two operating systems set up memory for a process in different ways, the above-mentioned areas needed by a PEPascal program at run-time will not always be located in the same relative location in the process' memory space. In OS/32, the UDL, the user program object, RTL objects, the stack, the heap, and other areas will be located primarily (and perhaps entirely) in the task's impure segment in the order shown in Figure 5. The user may specifically request certain of these areas to be in one of the other three segment types.

In UNIX, location #0 will always be the starting address of /lib/crt0.o rather than the UDL. Consequently, any locations in the UDL needed by PEPascal under UNIX will have to be stored elsewhere in the process' memory. The user program objects, PEPascal RTL objects, and the C library objects will make up the rest of the text segment. Any "PURE" object code is in this segment. The data segment must then hold the other areas such as the SDA, SCA, stack, and heap. Lastly, the stack segment must be used to communicate with C library routines. A map of these areas as they would appear in a UNIX process is shown in Figure 6. The issue in the port, then, is to allocate the memory and initialize all pointers to areas within it so that the OS environment is transparent to the compiler-generated user object code. This will prevent the need for wholesale changes in the compiler code-generation passes and thus help preserve its portability. The details of how this was implemented are found in section 3.2. below.

2.3. Error Handling

2.3.1. OS/32 interrupt handling

A third area of conceptual interest to the port is how system-detected errors are handled. Like most operating systems, the OS/32 kernel will handle (or "trap") internal interrupts from the processor [8]. For example, if the opcode of the next instruction is not in the processor's instruction set, i.e., an "illegal instruction," the processor will not try to execute it. Instead, it will "interrupt" the execution of the task. OS/32 then traps this interrupt so that it can gracefully (although sometimes cryptically) inform the operator/user of the problem. Left to its own, OS/32 will pause the task and display an appropriate message. After the operator or user has had an opportunity to check the process environment and perhaps correct the problem, the task may either be continued or killed. Some other examples of events which trigger interrupts are arithmetic faults, machine malfunctions, data format and alignment faults, and operator/user intervention. Some interrupts are fatal in that they do not give the user the option of attempting to continue the task.

Any custom error handling from user-programs would be next to impossible if the operating system always handled the interrupts. Therefore, OS/32 allows a user task to service the trapped interrupts. This feature is enabled by setting the appropriate bit in the TSW. Different traps are caught by setting different bits. This is one action which relies heavily on the task's UDL since the old and new TSWs are swapped in and out of this area. The handling routine is responsible for saving general and floating point registers before servicing the trap. After the service routine finishes, the old TSW is swapped back to continue normal execution, provided the service routine did not terminate the task. PEPascal uses this facility for handling run-time errors.

2.3.2. UNIX signals

The UNIX kernel can likewise handle interrupts caused by abnormal events. In UNIX, these are referred to as "signals" from the processor [18, 19]. One could classify these signals into 2 groups: signals from the outside world and program faults. The former consists of signals sent by the user or another process while the latter are normally hardware-generated faults such as an illegal instruction, arithmetic fault, or memory fault. The default action by UNIX upon catching these signals is to print a message and terminate the process. Some signals by default also dump the process' memory image into a file called "core" in the current directory. The user can then employ a debugger to investigate the state of the process when it died.

Like OS/32, the default response to a signal can be overridden with control passed to a handling routine. Rather than setting a bit in a TSW, default action in UNIX is altered with the "signal" system call. The C routine which accomplishes this expects as arguments the type number of the signal to be caught, followed by the address of the handler routine. It is up to the handling routine to do any register saves if needed. Unlike OS/32, though, control does not automatically return to the original process once the handler is done executing.

2.3.3. Relevance to the port

Error handling is primarily important in PEPascal for providing meaningful and graceful handling of run-time errors. If a user program attempts to divide by zero, for example, the processor will detect this, prevent the "divide" instruction from getting executed, and generate an arithmetic fault. Just a generic message indicating such a fault occurred and termination of the process would leave the programmer in a state of bewilderment. A message with the source code line number, address of the bad instruction, and type of error would be much more useful. Thus PEPascal has an error handling routine which gives this information to the user.

Since both OS/32 and UNIX allow user processes to override the default action and pass control to a user-written handler, the same handler routine can be used in the port. The important difference is how this feature is enabled. The UNIX port must make calls to the C routine "signal" rather than setting bits in the task's TSW and providing a new TSW in the UDL for the handler.

2.4. File Handling

The last area of interest in the porting process is file management. Not only are the PEPascal source code and object code in files, but Pascal allows the program to interact extensively with files. Files allow the user program to communicate with its external environment. They are a very important part of the PEPascal environment. Any process wishing to interact with files must request services from the host operating system in order to accomplish this. File handling in PEPascal is thus strongly dependent on the underlying OS.

2.4.1. OS/32 files

OS/32 supports two types of files: indexed and contiguous [8]. Indexed files are open-ended in that their size can increase or decrease dynamically. The only size limit is the physical space available. Contiguous files are a fixed-size structure; they can neither increase nor decrease in size. Their size is fixed when it is allocated. While the latter file type may waste space, it can ensure that space is available and facilitate fast random access of data. In most situations, though, the programmer will use indexed files since they do not waste space. OS/32 file management and file I/O is handled by SVC 7 and SVC 1 routines, respectively.

The internal structure of an OS/32 file is also determined when it is allocated. For indexed files, this structure is based on the "logical record." Although the length of a logical record in a file can be up to 65,535 bytes, it is commonly a value like 80, 120, 256, or 512 bytes. This length is fixed for a file once it is allocated. This is also the basic unit of data transfers between the physical file and the system buffers in the file's File Control Block (FCB). An SVC 1 call actually reads or writes individual logical records. Less than a full logical record can be read or written, but this wastes space since the I/O is still based on the entire logical record; the system pads the buffer to fill it up to the logical record length. Consequently, end-of-file condition is determined only to the nearest logical record.

2.4.2. UNIX files

By contrast, UNIX views files in a much more simplified, uniform manner. Instead of all the rigidly imposed structure of OS/32 files, UNIX just considers all ordinary files to be a one-dimensional array of bytes [1,19]. A file contains whatever the user puts in it. A text file is simply a series of "lines," where a line is a string of any number of characters terminated by a newline character (ASCII decimal 10). Binary (object) files are sequences of words as they will appear in core memory when executed. In essence, any structure imposed on a file is done by the program which uses it, not by the system.

The basic unit upon which physical I/O is based is a 512-byte block (as opposed to the OS/32 logical record). However, the system maintains I/O buffers which are in essence a data cache from which the user process requests data. If the requested location is not in the one of the blocks in the buffer, a buffer block is swapped out and the correct block is read in from the physical device. This is all transparent to the user for whom all I/O is based on reading/writing any number of bytes. In other words, there is no restriction to a particular record-size. As a result, end-of-file condition is determined to the nearest byte. The lowest level file handling and file I/O is done with UNIX system calls, but the C library provides a vast number of additional I/O routines [18].

2.4.3. Relevance to the port

So what does this mean to the UNIX port? First of all, the OS/32 file management and file I/O SVC calls will need to be translated into their respective C library calls (see Section 2.1.3.1.). This poses several problems. Identifying the file is one. The OS/32 file SVCs use either a file descriptor ([volume:]filename[.ext][/acct#]) of a "logical unit" (explained later) to identify the file. These are both stored in the SVC parameter block. The C library routines identify files in any one of three ways: the pathname, the address of the file control structure, or a small integer also called a file descriptor. The system calls all use the file descriptor so the UNIX port must keep a table of file descriptors for all open files.

Another problem in the SVC to UNIX system call translation arises from the different basic I/O unit as it appears to the programmer. OS/32 I/O is based on the logical record while UNIX I/O has a single byte as the fundamental unit. For non-text files, this causes little trouble since OS/32 will allow the user to read a single file component into the FCB buffer. The system may actually read more bytes than that to fill the last logical record, but this is transparent to the user. Text files are a different story. To OS/32, "read the next line" simply means go fetch the next logical record with an SVC 1. There is only one physical line stored per logical record. To UNIX, though, "read the next line" requires moving the text pointer to the first character past the next newline character in the file buffer. Only if the text pointer reaches the end of the buffer is the "read" system call used to fetch another buffer of characters from the physical file. As a result, the UNIX port will normally require fewer OS service calls per standard I/O routine call.

These file handling differences also mean that UNIX requires less file management overhead. For example, there is no need to "fetch the attributes" of a UNIX file since they can all be treated the same way. Allocating a file is also simpler since you do not need to bother with choosing the file type and structure. Finally, OS/32 allows the user to treat files and devices interchangeably by "assigning" the file or device to a "logical unit" [8]. A user task then interacts with the logical unit rather than directly with the file or device. In UNIX, files and devices are both implemented as files in the overall file system [18]. In essence, devices are just "special" files so that a uniform interface controls all communication between a user program and peripheral devices (as well as text files). More importantly, the physical differences are transparent to the user. Therefore, a user process may interact directly (in appearance, at least) with a file or device without the need for an intermediate generic logical unit.

3. IMPLEMENTATION

3.1. The Run-Time Library

3.1.1. Production environment

The PEPascal RTL consists of over 7000 lines of well-documented CAL code. The RTL routines provided by Perkin-Elmer were all in one file, set up to be assembled with the "BATCH" option. Since our version of the CAL assembler, `as', does not support this pseudo-op, each RTL routine was put into a separate file. These files were then grouped into subdirectories according to function. Figure 7 shows the hierarchy of the file system used in developing the UNIX version of the PEPascal system.

The UNIX utility `make' [20] was used to manage the many RTL source files and build the overall compile-time and run-time system. Appendix B contains the contents of the makefile used in the development. The makefile was set up such that the developer need only type `make' to re-assemble or re-compile any parts which have been updated more recently than its corresponding object file. Since the RTL routines rely heavily on PE's CAL Macro Processor utility [21], the makefile runs CAL files through CAL Macro before assembling them. C files are compiled with the "compile-only" option. All the RTL routines are then loaded into two library object files used by the link editor, `ld', to resolve external references in the compiled user-program.

The makefile also facilitates building CAL routines used to test individual RTL routines (explained in section 3.6), as well as the

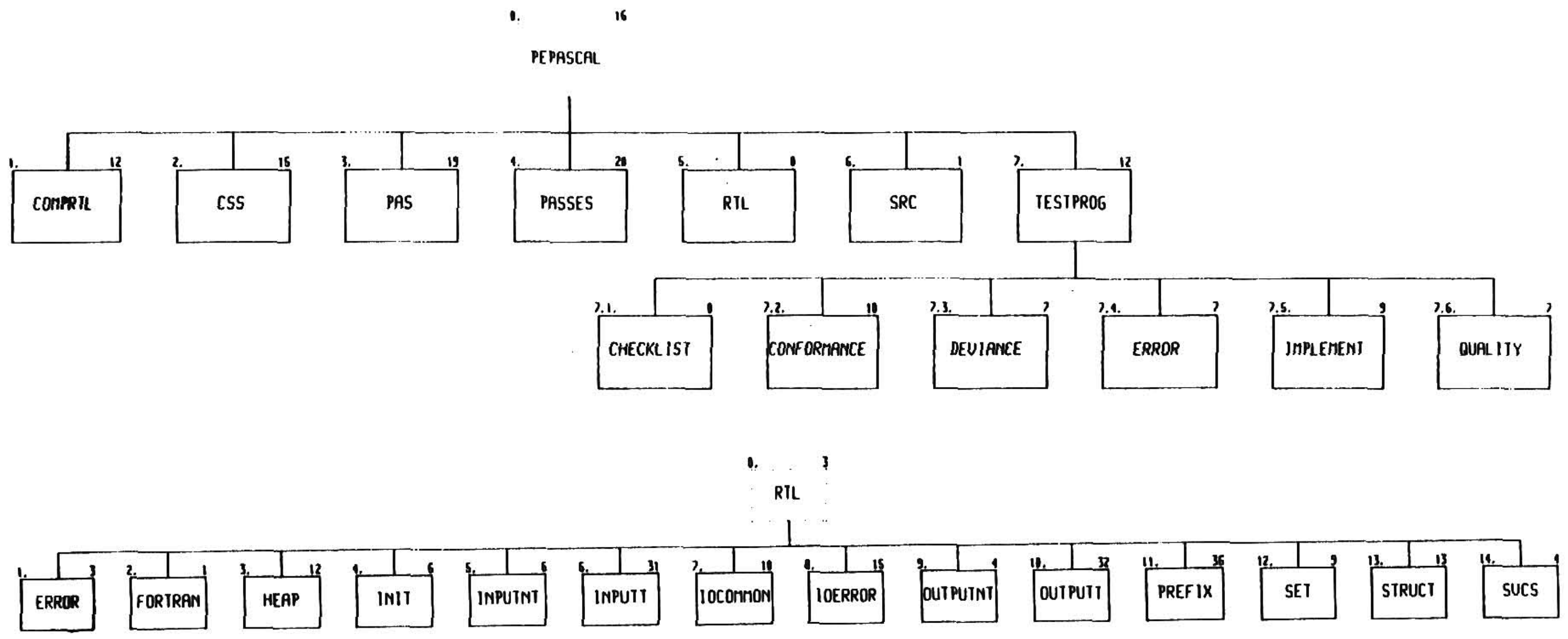


Figure 7. File hierarchy of the development system for PEPascal. The parent directory, pepascal, is in /usr/src. The boxes represent directories. The number of files in each directory is shown on the upper righthand corner of each directory.

public PEPascal system. The command `make test' does the former, `make public' the latter. Finally, the makefile builds the special run-time library, "comprtl", used by the compiler and the macro utility library, "mutil.lib".

As mentioned earlier, the RTL routines are written using the CAL Macro utility. The macros used in the RTL are in the file "macros.src". They provide such commonly used structures as the FCB, SVC parameter blocks, and register set mnemonics. It also is used for routine "ENTRY" and "LEAVE" operations which save specified registers across the routine call. This reliance on CAL Macro posed two options to us for porting the routines to UNIX. One option was to expand the RTL routines into CAL instructions with CAL Macro on a system running OS/32, then use the expanded code to integrate the UNIX port into. The other option was to port the CAL Macro facility to UNIX and use the RTL macros for the UNIX version also. The latter was the option chosen. During development, then, the RTL routines remained at a manageable size since the routines were only expanded immediately prior to assembly. Furthermore, I was able to use many of the provided macros in the code for the UNIX port. Since I also had to change some of the macros provided by PE, a utility to build macro libraries was developed.

Another decision in the porting process was whether or not to completely rewrite the RTL routines or to integrate the UNIX port into the OS/32 version. We chose the latter since it clearly would save programming time. It is also a better choice from the standpoint of simplifying maintenance for Perkin-Elmer; they only have one RTL to maintain. To accomplish this, I employed conditional CAL assembly. A macro called "options" which contains symbols that allow the user to define the target OS. If the symbols "unix" and "os32" are defined as 1 and 0, respectively, then the UNIX RTL is built. Object code for the OS/32 RTL is produced if "unix" = 0 and "os32" = 1. The "options" macro is defined in the file "macros.ksu".

3.1.2. Classification of RTL Routines

The PEPascal RTL is conceptually separated into six groups:

1) Initialization.
2) Error handler.
3) The RELIANCE - Pascal interface and error handler.
4) Pascal prefix support routines.
5) Pascal SVC support routines.
6) Pascal library support routines.

Since the SVC support is specific to an OS/32 environment, Perkin-Elmer did not wish to port this extension to the language. Likewise, the RELIANCE interface has no application in a UNIX environment and therefore was not ported. Many prefix routines are also OS/32 - specific. They too were not implemented. However, several prefix routines were needed to get the compiler running and are potentially useful in a UNIX environment. They were consequently ported. The other three RTL groups (initialization, error handling, and the library support routines), on the other hand, are all needed by the UNIX implementation. The implementation of these groups, as well as the prefix group, will be described in detail in the following sections.

3.2. Initialization Group

This group contains only one routine, P$INIT, which initializes the run-time environment of the compiled user program. In the production system, this routine is in the file RTL/init/p_init.s. In OS/32, P$INIT organizes the task work space described in section 2.2.3. Besides allocating the memory, it sets the pointers into this workspace, local base (LB), global base (GB), and stack limit (SL) (see Figure 5). It then initializes the contents of the allocated memory to zero. P$INIT also copies parameters from the OS/32 "START" command into a buffer occupying the top 132 bytes of the workspace (which is the bottom of the heap) so that they can be accessed from a program with the "START_PARMS" prefix routine. Finally, if specified when the OS/32 task is established, the single and/or double precision float registers are initialized to zero. All these actions and more were included in the ported P$INIT.

3.2.1. C stack

The first action that the UNIX version takes is to set up the C stack and initializes pointers to it (Figure 4). When control enters P$INIT, general register 7 (r7) points to the `a.out' command-line parameters. This address is preserved in the symbol "c.parms". The first word in c.parms is the count of how many arguments were on the command line (argc), followed by the addresses for each individual parameter (argv[]). P$INIT next subtracts 96 off of r7 to preserve 96 bytes for PEPascal to work with. This address is saved in "c.sp". This is considered by the PEPascal run-time environment to be the stack pointer into the C stack segment used to communicate values between Pascal and C routines. Thus, any Pascal routines needing to switch to C mode first loads "c.sp" into r7. It then saves all the registers on the C stack in the register-save area (Figure 4). The first 32 bytes above c.sp are then available for parameters passed to the C routine. To switch back to Pascal mode, the Pascal routine simply needs to restore the registers. This destroys r7 as the C stack pointer again.

3.2.2. Command-line parameters

The next action by the UNIX version is to parse the parameters given by the user on the command-line when the compiled program is invoked. P$INIT recognizes three types of parameters (see manual entry in Appendix C). Anything else is put on the bottom of the heap so that it is available to the user program just like the OS/32 "START" parameters. One possible option is "-d" (upper or lower case) which sets the flag "coreflag" in the global data area (gbdata) in the SDA. This flag is used by the error handler to determine whether or not to dump the core image upon detection of a run-time error. If the user includes the "-d" argument, the core image will get dumped. The default action is to not dump the core. Another recognized argument is "-kx" which alters the 8K default memory allocation for the Pascal workspace. "x" is the number of bytes to allocate. P$INIT recognizes the suffixes "k", "b", and "w" for multiplying the number by 1024, 512, and 4, respectively.

The third possible argument type deals with the assignment of external Pascal file identifiers with actual UNIX files. I mentioned in section 2.4.3. that UNIX file identification at the lowest level is based on the file descriptor while OS/32 uses the logical unit (LU). The PEPascal compiler also deals with external file identifiers as LU's. The first file in the program header file list is assigned to LU 0, the second file to LU 1, third to LU 2, and so on, up to a maximum of LU 31. I therefore built a table (fdtab) in the gbdata area which maps LU's into UNIX file descriptors and UNIX pathnames. The LU is the index into the table where the first fullword is the UNIX file descriptor and the second fullword is the address of the pathname. P$INIT initializes all the file descriptors to -1 and filename pointers to 0 (null). This is where the third argument type comes in. This argument-type associates a UNIX pathname with an LU. P$INIT takes an argument of the form "-fx filename" and saves the pointer to 'filename' in the fdtab entry for LU 'x'. This filename is then used by other RTL routines, such as P$RESET, P$RESETT, and P$REWRIT to open the file and get a UNIX file descriptor. Note that P$INIT only copies filename pointers into the fdtab. It does not verify that the file exists or is assigned to the correct LU. Other routines do these checks.

P$INIT does check for proper form of these three argument types. Any suffix other than "k", "b", or "w" with the "-kx" option produces an error. If the argument after "-fx" starts with a '-', an error is produced since P$INIT expects a filename. Likewise, a non-digit LU is an error. For any such error, P$INIT logs a message of the form

```
"pascal: invalid parameter <badparm>"
```

where the illegal argument is echoed. It then terminates the process so the user can try again.

3.2.3. Memory allocation and pointer initialization

The OS/32 version of P$INIT uses an SVC 2 code 2 "get storage" to allocate memory for the Pascal workspace. The UNIX version uses the C library routine "malloc" which expects as its parameter the number of bytes to be allocated. The OS/32 version relies on the compiler option MEMLIMIT to determine how much of this workspace is to be used for the Pascal stack and heap. A value of 100% allocates all of the memory for this purpose. The extra memory available if MEMLIMIT < 100% is used only if requested by an externally linked routine. Since UNIX can dynamically increase the size of the data segment, MEMLIMIT is not a meaningful option in a UNIX environment. If space for FORTRAN linkage is needed, it can be allocated with a call to "malloc". Thus, the user is advised to leave the MEMLIMIT option at its default value of 100%.

As in OS/32, the memory is allocated in two steps. First, memory for the SDA and the RTL scratchpad is allocated. "Malloc" returns the starting address of the memory allocated so this is used to set the pointer "udl.ext" which holds the address of the top of gbdata in the SDA. The memory for the remaining workspace is then allocated with a second call to "malloc". This is the area with a default size of 8K which can be altered with the "-k" parameter. The returned address is then used to initialize the pointers LB, GB, and SL. "Malloc" returns a zero (0) if there is not enough memory available to allocate the requested number of bytes. In this case, P$INIT prints the message

"Not enough space to run pascal"

and terminates the process. This is the same action as in the OS/32 version. The last action in P$INIT is to initialize the single and double precision floating point registers to zero. This is done automatically in the UNIX version whereas in the OS/32 version, this is only done if the FLOAT and/or DFLOAT options are specified when the task is established. The memory map of a PEPascal process in a UNIX environment upon exit from P$INIT is shown in Figure 6.

3.3. Error Handler

3.3.1. Handler initialization

Immediately after calling P$INIT, all PEPascal processes call the error handler initialization routine, P$ERR. In the OS/32 version, a new TSW is set up in the task's UDL for the error handler routine, PASERROR. It also sets the appropriate bit in the TSW to enable illegal instruction traps since this is the type of interrupt generated for run-time errors. An SVC 9 is used to swap TSW's.

The UNIX version initializes the error handling in a much different way. As mentioned in section 2.3.2., UNIX enables signal handling by calling the C library routine "signal". Before giving "signal" the address of the error handler, however, it is important to test whether or not a particular signal is currently set to be ignored. This is due to the possibility that the process is running in the background. If this is the case, the UNIX shell has set certain signals to be ignored so that when such interrupts occur, they do not kill this background process too. For example, if the user runs the PEPascal process in the background, starts another process, then decides to kill the foreground process with the DEL key ("interrupt" signal), the background process will be killed too unless it is set to ignore this signal. As a result, if P$ERR finds that a signal type is set to be ignored, it leaves it that way. If the signal is set for anything else (normally just the default action by the OS), then P$ERR calls "signal" again, this time passing the address of the error handler, PASERROR. Then, if any of these signals are detected, control passes to PASERROR.

One more issue is important in the error handling initialization. When a signal is trapped, UNIX puts the old PSW status, program counter, and stack pointer on the C stack before giving control to the error handler. UNIX expects r7 to point to the *top of the stack*. The problem is that in Pascal mode, PEPascal does not protect r7 as the C stack pointer. Thus, P$ERR must set the process to run in "no-stack" mode so that UNIX sets up a pointer 512 bytes off the top of the stack segment to save these values when a signal occurs. This is done in P$ERR by calling "signal" with the high-order bit of the PSW set.

3.3.2. The error handler

As mentioned earlier, control is given to PASERROR when a non-ignored signal is caught by UNIX while a PEPascal process is running. At this point in UNIX, r7 points to the location in the C stack segment where UNIX saved the environment. The first word on the C stack is the signal number caught. If a Pascal-detected *run-time error* occurred, then the signal will be number 4 for an "illegal instruction" because the compiler generates code that executes an "ERR" instruction (opcode #88) when such an error occurs. Pascal-detected errors have their own messages so if the illegal opcode was #88, then PASERROR must handle the signal differently. The opcode is the first byte of the old pc, so if it is #88, then PASERROR also retrieves the error code (2nd byte in pc) and the source line number (last halfword). The error code is then used as an index into a table of possible run-time error messages (Table 3). An error message of the form

"Line XXXX Address YYYYYY <message from table>"

is then sent to UNIX standard error (stderr) which defaults to the user terminal.

The global flag "coreflag" (sec. 3.2.2.) is then checked to determine whether or not to dump the core image before killing the process. The UNIX "quit" signal is used to dump the core while the RTL routine P$TERM terminates without a dump. Note that the "Quit" signal trap must be reset to its default action before using it to kill the process. Otherwise, UNIX will have PASERROR handle this signal too.

The other possible error condition is when any signal is caught other than the "illegal instruction" produced by the ERR instruction. These are UNIX-detected errors rather than Pascal-detected run-time errors. Some examples include "interrupt" or "quit" in response to user intervention, and "memory" or "address" faults from bad data. These errors produce the same message format shown above for Pascal run-time errors except that no source code line number is available and the error type message comes from a different table (Table 4). In this case, the line number is left at the default value of 0 making debugging particularly painful. Finally, the "coredump" flag is checked before terminating the process in the same way as described above.

Table 3. Perkin-Elmer Pascal run-time error messages. These strings are used by the run-time error handler, PASERROR, in response to Pascal-detected run-time errors. They are printed to UNIX stderr along with the source code line number and address of the instruction causing the error.

```
*
*       pascal-detected run-time error message table
*
        align adc
ertab   equ   *
        db    c'breakpoint      '
        db    c'index range err '
        db    c'param range err '
        db    c'value range err '
        db    c'case label error'
        db    c'trunc range err '
        db    c'variant tag err '
        db    c'pointer error   '
        db    c'stack overflow  '
        db    c'heap overflow   '
```

Table 4. Run-time error message text for UNIX signals. These strings are used by the run-time error handler, PASERROR, in response to UNIX-detected run-time errors. They are printed to UNIX stderr along with the default source code line number of 0 and the address of the instruction causing the error.

```
*
*       UNIX-detected signal message table
*
ersigtab equ   *         index by 16* signal number
     db   c'signal 0        '
     db   c'hangup          '
     db   c'interrupt       '
     db   c'quit            '
     db   c'illegal instruc '
     db   c'trace trap      '
     db   c'iot instr       '
     db   c'emt instr       '
     db   c'arith fault     '
     db   c'kill            '
     db   c'mem/align fault '
     db   c'address fault   '
     db   c'unix call error '
     db   c'pipe error      '
     db   c'alarm trap      '
     db   c'termination trap'
     db   c'signal 16       '
```

3.3.3. Utility routines

Three other routines are provided for run-time error handling. P$PAUS is an OS/32-dependent routine which pauses the user's task. This was not implemented in UNIX. Any call to P$PAUS simply returns to the caller. P$TERM is a utility which terminates the process. The OS/32 version used an SVC 3 to end the task after releasing storage with an SVC 2 code 3. UNIX automatically does all the memory housekeeping when a process terminates so the UNIX version simply calls the C library routine "exit" in P$TERM. OS/32 also uses an end-of-task code to indicate the condition under which the process is being terminated. In both operating systems, an exit code of 0 means normal termination, but the UNIX port simply maps all non-zero end-of-task codes into an exit code of 1 which is sent to "exit". It is thus possible for a parent process running a PEPascal process (e.g., the UNIX `sh' shell) to test for normal termination of the PEPascal program.

The third utility, P$SEND, is used to log a message. OS/32 does this with an SVC 2 code 7 "log message" call that prints the message on the system console. The UNIX version prints the message to stderr. It simply logs the message with a call to the C routine "write", then returns to the caller. A normal sequence for a routine using these utilities is to print the message with P$SEND, the kill the process with P$TERM.

3.4. Prefix Group

Only the prefix routines used by the compiler driver and passes were considered in this phase of the project. These routines, as well as all the prefix routines, are listed in Table 5. The decision on whether or not to port any of the other prefix routines was left to a later phase in the project.

A number of the prefix routines are only used by the compiler in special cases. OPEN and CLOSE, for example, are only used by passes 1 and 10 when the source program is on a non-random device, such as a magnetic tape. Pass 1 opens a temporary disk file to copy the source into so that pass 10 does not have to rewind the source device to re-read it for the output listing. Since this is a rarely-encountered condition, these routines were not ported. The routines WRITE_FILE_MARK, BREAKPOINT, and FORWARD_FILE_MARK fit this category too.

Five other prefix routines, TIME, DATE, EXIT, FETCH_ATTRIBUTES, and START_PARMS are unavoidably used by the compiler. TIME and DATE are only used by pass 10 for the output listing page headers. OS/32 provides this information with SVC 2 calls. I thus translated these into a call to the C routine "time" in both cases. The value returned by "time" is then submitted to the C routine "localtime" to get the appropriate values to generate the 8-character strings expected by the OS/32 TIME and DATE.

The FETCH_ATTRIBUTES prefix routine is also used only in passes 1 and 10. In OS/32, it uses an SVC 7 call to get the attributes of the file or device assigned to the logical unit specified in the parameter

Table 5. Perkin-Elmer Pascal prefix routine names as called from a Pascal program. The starred ('*') routines are used by the compiler. Some of these routines are only used in special conditions so they were not ported. The others were either fully moved to UNIX or altered just enough to get them to work with the compiler.

```
* open
* close
* allocate
  rename
  reprotect
  delete
  change_access_privileges
  checkpoint
* fetch_attributes
  rewind
* write_file_mark
  back_record
  back_file_mark
  forwd_record
* forwd_file_mark
* breakpoint
* start_parms
* time
* date
* exit
```

list. Since UNIX treats files and devices the same, there is no need for this request in UNIX. I consequently just changed the routine to return some standard UNIX-oriented values. Again, this was just enough to get the compiler to work and will not be a part of the final version of the ported language.

The START_PARMS routine is used by pass 10 to produce the table of compiler option settings printed in the output listing. As mentioned above in section 3.2.2., legal command-line parameters not used by P$INIT are put at the bottom of the Pascal heap. START_PARMS then just reads these 132 bytes into a character array. Therefore, no changes were needed to have this routine work in UNIX. Likewise, the routine EXIT was not changed since to terminate the process, it just calls P$TERM which was ported as described in section 3.3.3. above.

Several of the prefix routines presented the problem of name conflicts. When the compiler code makes a call to OPEN, for example, what prevents branching to the C routine "open" rather than the RTL prefix routine of the same name? Only the prefix routines OPEN, CLOSE, EXIT, and TIME had this potential problem. Since these four routines were not crucial to the compiler, I was able to get the compiler working crudely without dealing directly with the problem. Once I could recompile the driver and passes, though, I changed the entry labels to POPEN, PCLOSE, PTIME, and PEXIT so that they would not conflict. The other RTL routines which one would expect to have the problem, such as READ and WRITE, do not conflict with C routines because their entry labels start with the prefix "P$".

3.5. Library Support Group

This group of routines performs the detailed functions for many of the PEPascal language features. This includes heap management, manipulation of structured variables, set operations, and file I/O. Since the former 3 functional subgroups of the library support routines are handled without the aid of any OS services, they needed no changes. The source for these routines are in the subdirectories "heap", "struct", and "set" as shown in Figure 7. The latter subgroup, file I/O, relies heavily on OS services. Consequently, the major effort in porting the library support routines dealt with file I/O. These routines are in the subdirectories "inputnt", "inputt", "outputnt", "outputt", "iocommon", and "ioerror".

3.5.1. Non-text file input

The subdirectory "inputnt" contains the source for the routines P$READ, P$GET, and P$RESET. P$READ needed no changes since it calls P$GET to do the actual I/O. P$GET, then, simply fetches a single file component into the file's FCB buffer. In OS/32, this means "read the next logical record" with an SVC 1 call (see section 2.4.3.). The porting process for this routine was thus primarily translating the SVC 1 into a call to the C routine "read". To execute the UNIX "read", P$GET must determine the exact number of bytes for the given file component. This size is calculated from the starting and ending addresses of the buffer. Two special error conditions had to be dealt with, too. For one, this routine prints a message when UNIX has a problem reading from the file (i.e., when "read" returns -1). This can happen if the file does not exist or does not have read permission, for example. Secondly, P$GET prints a message and terminates the process if UNIX was not able to read all the bytes requested, i.e., the entire file component.

P$RESET initializes a non-text file for reading. This routine is called in response to the code "reset (fileid);" in the Pascal program. In OS/32, this simply involves an SVC 7 "fetch attributes" call since the file is assigned to its LU before the program is invoked. If the file supports it, the file is then rewound through a call the P$$REWD. P$RESET finally calls P$GET to read a file component into the buffer. The UNIX version basically replaces the "fetch attributes" call with a call to the C routine "open". If the file is already open, it is first closed before being reopened with read-only permission. This can happen if the file had been written to earlier in the program.

The UNIX version handles two error conditions in P$RESET. If there is no file name saved in the global fdtab for the LU being opened, a message of the form

```
"reset: logical unit X unassigned"
```

is printed. This normally happens when the user forgets to assign a UNIX file to the LU in the command-line. The other error occurs when UNIX has a problem opening the file. This can happen, for example, if the user does not have read permission on the file or the file does not exist. P$RESET just calls the C routine "perror", passing to it the bad file name, to print the appropriate message. The source for P$RESET is included in Appendix A.

3.5.2. Text file input

The routines in the "inputt" subdirectory perform many of the same functions for text files as the routines described above do for non-text files. The primary difference, though, is that everything is input as ASCII characters. Items such as integers and reals must therefore be converted from ASCII strings to their corresponding numeric value. However, these conversions are all done internally without any OS services. Therefore, only the routines which perform the actual I/O transfers needed to be changed. These routines were P$READLN, P$GETT, and P$RESETT.

Text files in PEPascal are managed differently from non-text files. Non-text files simply read another file component into the file's FCB buffer when the next value is requested. Since a text file's component is a single character, that same strategy would be very inefficient. Therefore, PEPascal maintains a buffer of 256 characters in the text file's FCB. As a result, most text file I/O relies on P$GETT to move a pointer to the next character in the FCB buffer.

Only two changes were needed for P$GETT since it just manages an internal buffer; it calls P$READLN to do the actual I/O transfer, if needed. For one, OS/32 uses the carriage return character (ASCII decimal 13) to denote the end-of-line while UNIX uses the newline character (ASCII decimal 10). Secondly, the OS/32 version forces an end-of-line condition if the end of the buffer was reached since a single buffer represents a logical record which is a "line". The UNIX version skips this and calls P$READLN to get the next buffer of characters. P$READLN is only called when either the text pointer gets to the end of the buffer or a new "line" is requested.

As stated in section 2.4.3., a request for a new line means two different things in OS/32 and UNIX. In OS/32, "read a new line" means simply to fetch the next logical record from the file with an SVC 1 call. In UNIX, this means to move the text pointer to the first character past the next newline character in the buffer. This basic difference forced me to completely rewrite the algorithm for P$READLN. The new algorithm is included in the routine's comment header listed in Appendix A.

The condition which caused the most trouble in porting P$READLN was when the next newline character fell near the FCB buffer boundary. If, for example, the next newline in the file is the last byte in the buffer, P$READLN must call the C routine "read" to get the next 256 bytes, the reset the text pointer to the first character in this new buffer before returning to the caller. P$READLN therefore has to remember whether or not to continue searching this new buffer for a newline character. This is the purpose of the "recflag" variable. If it is clear, then the newline character was already found in the old buffer and there is no need to continue searching. Further complicating matters was the need to maintain the end-of-file and end-of-line flags. If the first character after the next newline happens to also be a newline character, then the end-of-line flag has to be set before returning. P$READLN was extensively tested because of all the off-by-one errors that could occur when dealing with the buffer boundary.

Like P$RESET for non-text files, P$RESETT for text files opens the file for reading. The UNIX version of P$RESETT thus closes the file if it is already open, opens it read-only, and fetches the first buffer of characters by calling P$READLN. The error messages are the same as for P$RESET.

3.5.3. Non-text file output

The subdirectory "outputnt" has only two routines for non-text file output: P$PUT and P$WRITE. P$WRITE simply maintains the internal FCB buffer. It copies the value of the file component into the buffer, then calls P$PUT to have the buffer dumped to the physical file. It therefore does not use any OS services and needed no changes to work in UNIX.

P$PUT, on the other hand, performs the actual I/O transfer so it needed changing. This primarily involved translating the OS/32 SVC 1 "write" to a call to the C routine "write". This required determining the number of bytes to be written to the file. This was done by subtracting the address of the start of the FCB buffer from the buffer end address. Two possible error conditions are handled. The C routine "perror" prints a message for a UNIX-detected error in writing to the file. For example, this can happen if the file does not exist, the user does not have write permission on the file, or the device experiences a hardware failure. The other error occurs if the entire file component was not written into the file. This again is most likely due to a hardware failure.

3.5.4. Text file output

The files in subdirectory "outputt" perform the output functions for text files. Since all output to these files is in ASCII characters, numeric values such as integers and reals must be converted to their corresponding ASCII string before the output is performed. This is the opposite conversion performed by text file input routines.

Like text file input, these conversions are mostly done without the aid of OS services. There is one exception, though. The conversion of an integer to its corresponding ASCII string is done in the OS/32 version with the help of an SVC 2 code 6. Since UNIX does not have a system call that does anything like this, the UNIX version of the routines that use this simply performs the conversion within the routine. This is done the same way for all three integer-writing routines, P$WRITBY (BYTE type), P$WRITSI (SHORTINTEGER type), and P$WRITI (INTEGER type). The source code for all three routines is in the file "pwrt.int.s."

Again, only a few routines perform the actual I/O transfers for text file output. The only two routines which needed to be changed were P$PURGE and P$WRITLN. Both routines flush the FCB buffer out to the file. Porting these routines involved translating an OS/32 SVC 1 "write" to a call to the C routine "write". The only difference between the two routines is that P$WRITLN appends a newline character to the end of the text in the buffer if the buffer is not full. This is necessary because P$WRITLN can be called for two different purposes. It can be called by P$PUTT to flush out a full buffer or it can be called directly by the compiled user program. In the latter case, it is used to designate the end of a line which in UNIX means "write a newline character" to the file.

P$PURGE and P$WRITLN handle the same two error conditions. Again, "perror" is used to print UNIX-detected errors such as the file not existing. The other error occurs when not all the characters in the buffer get written to the file. This normally indicates a hardware problem. The source for P$WRITLN is included in Appendix A.

3.5.5. Common I/O routines

Five routines perform functions which are common to both text and non-text files. They all reside in the subdirectory "iocommon." They are P$REWRIT, P$IFCB, P$EFCB, P$CLOSE, and P$$REWD. Only P$EFCB did not need changes since it is the only routine which does not use OS services.

P$REWRIT initializes a file for writing. It is called in response to the code "rewrite (fileid)" in the Pascal program. Like resetting a file, the OS/32 version simply uses an SVC 7 "fetch attributes" call to find out about the file assigned to the given LU. Since rewriting a Pascal file destroys the file, the UNIX version creates the file with the C routine "creat." This either creates the file if it does not already exist or truncates it to zero-length if it does exist. The UNIX version then closes the file and opens it write-only. After some OS-independent FCB initializations, the file is ready for writing.

Like P$RESET and P$RESETT, there are two possible errors. If the user did not assign in the command-line a UNIX file name to the LU, the following message is printed:

"rewrite: logical unit X unassigned"

The C routine "perror" is also used to print an error message if UNIX has a problem creating or opening the file.

P$IFCB sets up an internal file. It both creates the file and initializes its FCB. Porting this routine primarily involved translating an SVC 7 "allocate and assign" call to its UNIX-equivalent. The UNIX version uses the C routine "mktemp" to make a unique file name based on the process' identification number. This file, which is created in the directory "/tmp", is then created with "creat". The temporary file name and the returned UNIX file descriptor are then saved in the global fdtab so that P$RESET, P$RESETT, and P$REWRIT can reopen the file according to how it is used in the program. "Perror" is again used to print a message if UNIX has problems creating the file.

P$CLOSE is used to close an internal file created by P$IFCB. The OS/32 version simply does this with an SVC 7 "close" call. To port this routine, then, I translated the SVC 7 into a call to the C routine "close". The UNIX version also calls the C routine "unlink" which removes the temporary file from the file system. It also clears the fdtab entries for this file so they can be reused by another temporary file, if needed. If UNIX has a problem closing the file, then a message of the form:

"error in attempting to close an internal file"

is printed, followed by the UNIX error message printed by "perror".

The last routine ported in this group is P$$REWD. It is called by P$RESET, P$RESETT, and P$REWRIT to rewind the file or device assigned to the given LU. OS/32 does this with an SVC 1 "rewind" call. This was translated into a call to the C routine "lseek" which simply moves a file's file pointer to the beginning of the file. This is done to ensure that any read or write session for that file always starts at the beginning of the file. The only error condition occurs when UNIX has a problem during "lseek". In this case, "perror" is again used to print the message before the process is terminated.

3.5.6. I/O error servicing routines

There are seven routines used by other RTL I/O routines to handle special error conditions. Three of them, P$FCBERR, P$$SVC1, and P$$SVC7 are strictly OS/32-dependent. They were, therefore, not ported to the UNIX system. The other four, P$GETERR1, P$GETERR2, P$PUTERR, and P$NUMERR needed to be changed to work on UNIX for two reasons. For one, they all used an SVC 2 code 18 which moves ASCII characters between buffers in memory and an SVC 2 code 6 to convert the LU to an ASCII string. Secondly, a message in UNIX should have a newline character on the end of it so that any subsequent terminal output starts printing on the next line.

To port these four routines to UNIX, then, a message without the SVC calls and with a newline on the end had to be created. This was done with the C routine "sprintf" which creates a string from any combination of other strings, characters, and/or numeric values. The format of the messages was slightly changed, too. The UNIX version prints the name of the file which had the error rather than the number of the LU it was assigned to. In all cases and for both the OS/32 and

UNIX versions, the message is then sent to the terminal with P$SEND before the process is terminated. The source for P$NUMERR is included in Appendix A as an example.

3.6. Testing

To test the UNIX port of PEPascal, I first wrote CAL programs which simulated compiler-generated code to call each routine individually. This modular approach was not only a desirable strategy, but also a necessary one. It was desirable from the standpoint of easily isolating individual routines for testing. In the context of a compiled, linked user program, RTL routines are often embedded in a long, and in some cases difficult to predict, sequence of routine calls. It can thus be difficult to distinguish the effects of an individual routine. This strategy was likewise necessary since we did not have a working compiler. The RTL routines had to be working before the compiler could successfully compile a program which had calls to the RTL routines of interest from a high-level Pascal program. Once the compiler was working, however, I did test the RTL routines from PEPascal programs. I will discuss this in more detail later. The UNIX assembly-level debugger utility `adb' [22] was used extensively throughout the testing process.

The testing strategy used was the "path analysis testing" approach [23,24]. Since executing every possible path from entry to exit was not a practical possibility for most routines, test data was selected so that every instruction in each RTL routine was executed at least once

[23]. The modular design of the RTL aided this effort since each routine was kept relatively small. The number of paths through each routine was thus kept at a manageable level. In fact, some routines had only one path to test. Others, like P$READLN, were quite complex so I used a flow chart to help select paths to test. This effort also included testing error conditions. For example, I provided the RTL routine P$INIT with a variety of invalid parameters to make sure they were all detected and echoed in the error message.

Since every program first calls P$INIT for initialization of the run-time environment, I first tested this RTL routine. All programs likewise call the error-handling initialization routine, P$ERR, immediately after calling P$INIT. I thus tested this next. These two routines were used in all subsequent tests. File I/O routines also required initializing file control blocks for the files of interest so P$IFCB and P$EFCB were tested early, as were P$RESETT, p$RESET, and P$REWRIT. P$TERM was used to provide a controlled exit point from the test program since it exits with the message "Ending execution." These routines thus provided the background in which other RTL routines could be tested. As an example, Figure 8 shows the CAL code used to test the routine P$WRITLN which writes a line to a text file. In the interest of programmer efficiency, testing the routines in a wider context (i.e., in relation to a number of other RTL routines in addition to the minimum number described above) was deferred to when we had a working compiler since it is much easier to write Pascal code.

Once all the RTL routines had passed the above testing procedure, the compiler was loaded with the ported RTL object modules. This provided an immediate test of the RTL routines in the context of a

```
* file control block
*
fcb      struc
fcb.mw   ds    4                    flags (see below)
fcb.tptr ds    4                    text pointer, a(next_char)
fcb.cfsz ds    4                    current file size
fcb.svc1 ds    svc1.                svc 1 parameter block
fcb.svc2 ds    8                    svc 2 parameter block
fcb.svc7 ds    svc7.                svc 7 parameter block
         ds    80-*                 reserved
fcb.bufr equ   *                    buffer starts here
         ends
*
tfcb     ds    fcb
         ds    256              256 byte text file buffer
testprog equ  *
     lhi  14,#64                100% memlimit option
     lis  13,5                  min.lu
     bal  15,P$INIT                 run-time initialization
     bal  15,P$ERR              error-handling initialization
     lis  12,0                  logical unit
     lhi  13,256                size of textfile buffer
     la   14,tfcb               addr of file control block
     bal  15,P$EFCB                 init the fcb (external file)
     bal  15,P$REWRIT           set the file for writing
     bal  15,P$WRITLN           output the line to the file
     lis  14,0                  normal exit status
     b    P$TERM                exit w/o return
     end
```

Figure 8. The CAL program used to test the P$WRITLN rtl routine which outputs a line to the file assigned to logical unit 0. At this point, the rtl routines P$INIT, P$ERR, P$EFCB, P$REWRIT, and P$TERM had already been tested.

number of very large PEPascal programs. Pass 10 alone, for example, contains over 4,000 lines of PEPascal source code. A number of the RTL routines had to be modified at this point to get the compiler to work properly. Once the compiler was functional, however, testing then shifted to PEPascal source code programs.

In this phase of testing, RTL modules were tested individually, but this time in the context of a high-level PEPascal program. This phase concentrated more on testing special cases which would have been difficult and/or time-consuming to code in CAL. For an extreme example, consider the text file input routine P$READLN. As mentioned earlier, this routine must move the FCB text pointer to the first character past the next newline character in the FCB buffer. Understanding that the buffer occupies 256 bytes, I tested a large number of cases which read and wrote values right around the buffer boundary. Keep in mind that P$READLN must read another buffer from the file if the text pointer reaches the end of the current buffer and that it must pay attention to end-of-file and end-of-line conditions. For example, if the first character past the next newline in the buffer is also a newline (which would appear as a blank line in the input file), this routine must set the end-of-line condition to true before returning. This is complicated further if this exists across a buffer boundary.

The final phase of testing involved running the compiler through a Pascal Validation Suite [25] of test programs. This set of over 300 Pascal programs not only tested for conformance to the Pascal standard [3], but also a wide variety of situations including error handling, implementation-dependent features, and extensions to the language standard. This phase was very tedious and time-consuming since each test program had to be separately compiled and executed. However, it proved valuable in finding a number of obscure error conditions which I would have missed otherwise and increasing my confidence in the implementation.

3.7 Documentation

The primary emphasis in documentation was on internal documentation. The OS/32 version of the RTL routines provided by Perkin-Elmer was an excellent example of low-level documentation. As the examples in Appendix A indicate, each routine has a header as well as extensive in-line comments with the CAL instructions. The header consists of information about the interface with the calling routine, which registers hold what values when called, the action performed by the routine, where it is called from, and which routines, if any, it calls. My only addition to the header was to describe the action taken by the UNIX port of the routine where it differed from the OS/32 version and to show which routines the UNIX port called. In the examples in Appendix A, the lines added for the UNIX port are shown with a '¦' in the rightmost column. Note that in only one case, P$READLN, was the action by the UNIX routine sufficiently different to warrant completely rewriting the algorithm. In all cases, nearly every CAL instruction I added was abundantly documented with comments to clearly describe the local action.

External documentation consists of two on-line UNIX manual entries: one for the compiler and one for the run-time environment. The entry which describes how to use the compiler and all its options is invoked by the command `man pascal'. The entry for the run-time environment is obtained with `man 7 pascal' since it resides in section 7 of the on-line manual documentation and has the same name as the compiler description. This latter document includes a description of how to assign UNIX file names with the external files listed in the program header. These manual entries are included in Appendix C. Finally, just entering the command `pascal' will produce a brief synopsis of how to invoke the compiler with its options. This documentation, along with the PEPascal manual [7], provides the user with a very informative environment for compiling and executing their PEPascal programs.

4. SUMMARY

At the completion of this project, we had a working PEPascal compiler available to students and faculty. It had successfully recompiled its passes as well as a number of other programs written by students in the Department. This project was only one phase of the overall effort to port Perkin-Elmer Pascal onto UNIX, however. Other phases include modifying the compiler to produce native UNIX object code, providing an interface to permit calling C routines directly from the Pascal program, and writing a driver in C to coordinate the compilation and link-editing processes. Producing a functioning compiler, however, was the important first step.

The entire PEPascal system (shell and object code only) available to users on UNIX required approximately 1.25 Megabytes of disk storage. The development system which included all the source code for the compiler passes and the rtl occupied about 6.25 Megabytes. The compiler required 128K of main memory to compile a small to medium sized program. It unfortunately took as much as 200K to compile some of the passes which were over 4000 lines long. In terms of performance, the ported PEPascal does well once the user program is compiled and linked. However, any attempt to compile a large program when the host system has a heavy load will result in user anxiety - it is slow!

The project involved approximately 3 man-months of effort. This is a reasonable time frame when compared to other language ports [10,11,14] although the other implementations involved porting to a new machine. This is especially favorable considering the 3-year effort to implement a language from scratch on a new machine [11]. An important point to consider, though, is that this time was just for porting the rtl - very little was done to the compiler itself. In fact, the compiler still produces OS/32 object code. It still has to be converted to UNIX object with `cvobj'. Considering my previous lack of experience with assembly language programming, the run-time environment of languages, and the OS/32 operating system, I think 3 man-months is a strong testimony to the portability of the language implementation. Isolating the system-dependent features in a modular rtl proved to be a successful strategy for language portability.

REFERENCES

[1] Ritchie, D.M. and K. Thompson. The UNIX Time-sharing System. Comm. of ACM 26,1 (Jan 1983), p. 84-89.

[2] Wirth, N. The Programming Language Pascal. Acta Informatica 1 (1971), p. 34-64.

[3] ANSI and IEEE. _An American National Standard IEEE Standard Pascal Computer Language_. IEEE, New York, NY. (1983).

[4] Joy, W.N., S.L. Graham, and C.B. Haley. _Berkeley Pascal User's Manual - Version 2.0_. Univ. of Calif. at Berkeley Computer Center Library, Berkeley, CA (1980).

[5] Watson, J.A. SVS Pascal in the UNIX Environment. J. Pascal, Ada, & Modula-2 3,2 (Mar/Apr 1984), p. 25-28.

[6] Watson, J.A. Pascal and the UNIX Operating System. J. Pascal and Ada 2,6 (Nov/Dec 1983), p. 27-28.

[7] Perkin-Elmer Corp., _Pascal User Guide, Language Reference, and Run Time Support Reference Manual_, Pub. no. 48-021R01 (1982).

[8] Perkin-Elmer Corp. _OS/32 Programmer Reference Manual_. Pub. no. S29-613R04 (1979).

[9] Perkin-Elmer Corp., _Common Assembly Language (CAL) Programming Reference Manual_, Pub. no. S29-640R04 (1980).

[10] Poole, P.C. and W.M. Waite. Portability and adaptability, in F.L. Bauer, ed., _Advanced Course in Software Engineering_, Springer-Verlag, Berlin (1973).

[11] Brown, P.J. Macro processors, in P.J. Brown, ed., _Software Portability_. Cambridge Univ. Press, Cambridge, England (1977), p. 89-105.

[12] Griswold, R.E. _The Macro Implementation of SNOBOL4_. Freeman, San Francisco, CA (1972).

[13] Brinch Hansen, P. The programming language concurrent Pascal, IEEE Trans. on Software Engineering, SE-1, no. 2 (1975), p. 199-207.

[14] Neal, D. and V. Wallentine. Experiences with the portability of Concurrent Pascal. Software - Practice and Experience, vol. 8 (1978), p. 341-353.

[15] Norwegian Computing Center. SCALA, System Construction and Application Languages: S-PORT, the development of a Portable Simula System. Norwegian Computing Center, Forskningsveien 1B, Blindern, Oslo, Norway.

[16] Kernighan, B.W. and D.M. Ritchie. The C Programming Language. Prentice-Hall, New Jersey, (1978).

[17] Bell Telephone Laboratories. UNIX Programmer's Manual, Seventh Ed., Vol. 1, Sec. 2. Bell Telephone Laboratories, Inc., Murray Hill, NJ. (1979).

[18] Kernighan, B.W. and D.M. Ritchie. UNIX programming - second edition. In UNIX Programmer's Manual, Seventh Ed., Vol. 2a. Bell Telephone Laboratories, Inc., Murray Hill, NJ. (1979).

[19] Thompson, K. UNIX implementation. In UNIX Time-sharing System: UNIX Programmer's Manual, Seventh Ed., Vol. 2b. Bell Telephone Laboratories, Inc., Murray Hill, NJ. (1979).

[20] Feldman, S.I. Make - A program for maintaining computer programs. In UNIX Programmer's Manual, Seventh Ed., Vol. 2a. Bell Telephone Laboratories, Inc., Murray Hill, NJ. (1979).

[21] Perkin-Elmer Corp. CAL Macro Processor and Macro Library Utility. Pub. no. S29-408R04 (1979).

[22] Maranzano, J.F. and S.R. Bourne. A tutorial introduction to ADB. In UNIX Programmer's Manual, Seventh Ed., Vol. 2a. Bell Telephone Laboratories, Inc., Murray Hill, NJ. (1979).

[23] Beizer, B. Software Testing Techniques. Van Nostrand Reinhold Co., Inc. New York, NY. (1983).

[24] Howden, W.E. Reliability of the path analysis testing strategy. IEEE Trans. on Software Engineering SE-2,3 (Sept 1976), p. 208-215.

[25] Sale, A.H.J. and B.A. Wichmann. Pascal Validation Suite. A tape distributed by C/-Software Consulting Services, Allentown, PA.

Appendix A. Examples of ported Perkin-Elmer Pascal run-time library routines. They are written in CAL with the support of the CAL Macro processor. The symbols "unix" and "os32" are defined in the macro called "options". These symbols determine whether the UNIX or OS/32 run-time libraries are assembled. All code added in the UNIX port has a '|' in the righthand column.

```
**P$RESET
P$RESET  P.HEADR 01,00
         SPACE 5
* 7.5.3 PROCEDURE P$RESET(VAR F: UNIV FILE);
*
* INTERFACE:
*        THE ADDRESS OF THE FILE'S FCB IS RECEIVED IN R14.
*
* OS32 ACTION:
*        1. FETCH ATTRIBUTES OF THE LOGICAL UNIT.
*        2. IF THE FILE/DEVICE SUPPORTS BINARY I/O, SET THE SVC 1
*           FUNCTION CODE TO READ BINARY & WAIT; OTHERWISE SET IT TO
*           READ ASCII & WAIT.
*        3. IF THE FILE/DEVICE SUPPORTS REWIND, REWIND IT.
*        4. RESET THE FILE SIZE IN THE FCB TO ZERO.
*        5. SET THE STATUS FLAGS TO ALLOW INPUT AND SET EOF TO FALSE
*           FOR THE BENEFIT OF P$GET.
*        6. CALL P$GET TO READ THE FIRST RECORD.
*
* UNIX ACTION:
*        1. IF THE FILE IS ALREADY OPEN, CLOSE IT. THEN OPEN IT        |
*           READ-ONLY, SAVING THE RETURNED FILE DESCRIPTOR.            |
*        1b. CHECK FOR ERRORS IN THE OPENING OF THE FILE.              |
*        2. SAME.                                                      |
*        3. REWIND (WITH LSEEK) THE FILE.                              |
*        4-6. SAME                                                     |
*
* ERROR RESPONSES:
*    OS32: IF THE FETCH ATTRIBUTES CALL FAILS, A MESSAGE IS LOGGED     |
*        AND THE TASK IS PAUSED.  ON CONTINUATION, THE OPERATION
*        IS RETRIED.
*    UNIX: IF THE FILENAME PTR. IS NULL, THEN NO UNIX FILE HAS BEEN    |
*        ASSIGNED - LOG THE ERROR MESSAGE WITH P$SEND.  IF UNIX DETECTS |
*        AN ERROR IN OPENING THE FILE, LOG THE MESSAGE WITH 'PERROR'.  |
*
* CALLED FROM:
*        COMPILER GENERATED USER CODE.
*
* CALLS TO:
*    OS32: P$FCBERR, P$PAUS,  P$$REWD, P$GET                           |
*    UNIX: close, open, perror, sprintf, strlen, P$SEND, P$TERM,       |
*          P$$REWD, P$GET                                              |
        SPACE 5
        $PREGS LIST=NO
        $PASFCB LIST=NO
        TITLE P$RESET - NON-TEXT FILE INITIALIZATION FOR READ
        P.DATA
        P.SAVEM  R12
        ENDS
* Macros needed for UNIX                                                    |
        options                                                             |
        gbdata                                                              |
        udl                                                                 |
        SPACE 2
P$RESET P.ENTER
        SPACE 2
* 1.
RESET.0 EQU   *
        ifnz  unix                                                          |
        l     r7,c.sp               get c stack ptr                         |
        stm   r0,32(r7)             save pascal regs                        |
        space                                                               |
        lr    r11,r14               save the fcb address                    |
        lb    r12,fcb.svc7+svc7.lu(r11) get lu                              |
        sla   r12,3                 calculate fdtab offset                  |
        l     r13,udl.ext           get to top of gbdata                    |
        si    r13,gbdata            then get to the beginning of it         |
        ar    r12,r13               and calculate actual offset             |
        l     r13,fdtab+fd(r12)     get filedes                             |
        cli   r13,-1                is it unassigned?                       |
        be    openit                yes - open it w/o close                 |
        space                                                               |
*   close the file if it is already open                                    |
        st    r13,0(r7)             otherwise put it on c stack             |
        bal.ext llink,close         and close it                            |
        space                                                               |
*   open the file read-only if it has been assigned                         |
openit  l     r14,fdtab+fname(r12) get filename ptr                         |
        bz    filerr                error if null ptr                       |
        lis   r15,0                 read-only mode                          |
        stm   r14,0(r7)             put them on c stack for open            |
        bal.ext llink,open                                                  |
        st    r0,fdtab+fd(r12)      save the filedes                        |
* 1b.                                                                       |
        bm    openerr               error if negative fd                    |
        space                                                               |
        lm    r0,32(r7)             restore pascal regs                     |
        else  unix                                                          |
        LI    R12,0                 SET UP FOR FETCH ATTRIBUTES
        STH   R12,FCB.SVC7+SVC7.OPT(R14)
        SVC   7,FCB.SVC7(R14)       DO THE FETCH
        LB    R12,FCB.SVC7+SVC7.STA(R14)
        LR    R12,R12               FETCH OK ?
        BZ    RESET.1               YES, CONTINUE
        SPACE
```

```
         BAL.EXT LLINK,P$FCBERR     LOG ERROR MESSAGE
         BAL.EXT LLINK,P$PAUS       WAIT FOR CORRECTION
         B     RESET.0              RETRY
         endc                                                          |
RESET.1  EQU   *
         SPACE
* 2.
         LI    R13,S1FC.RDM+S1FC.WTM ASSUME ASCII+WAIT
         LH    R12,FCB.SVC7+SVC7.KYS(R14) GET ATTR FLAGS
         THI   R12,X'1000'          BINARY SUPPORTED ?
         BZ    RESET.2              NO, SKIP IT
         OHI   R13,S1FC.BIM         SET BINARY BIT
RESET.2  EQU   *
         STB   R13,FCB.SVC1+SVC1.FC(R14)  SET FN CODE
         SPACE
* 3.
         ifnz  os32                 assume unix files support rewind   |
         THI   R12,X'0040'          REWIND SUPPORTED ?
         BZ    RESET.3              NO, DON'T TRY IT
         endc                                                          |
         BAL.EXT LLINK,P$$REWD      REWIND THE FILE/DEVICE
RESET.3  EQU   *
         SPACE
* 4.
         LI    R12,0                RESET FILE_SIZE TO ZERO
         ST    R12,FCB.CFSZ(R14)
         SPACE
* 5.
         LI    R12,MW.RESET         FLAG FILE AS RESET
         ST    R12,FCB.MW(R14)
         SPACE
* 6.
         BAL.EXT  LLINK,P$GET
         P.LEAVE
         SPACE
* message for UNIX-detected error in opening the file                  |
openerr  equ   *                                                       |
         l     r14,fdtab+fname(r12) get filename ptr                   |
         sta   r14,0(r7)            put on c stack for perror          |
         bal.ext llink,perror       write the message                  |
         b     quit                                                    |
         space                                                         |
* error message for failing to assign a UNIX file to the pascal file   |
filerr   equ   *                                                       |
         la    r13,msgbuf           get address of buffer for message  |
         la    r14,errmsg           format string for sprintf          |
         lb    r15,fcb.svc7+svc7.lu(r11)  get unassigned lu            |
         stm   r13,0(r7)            put parms on c stack for sprintf   |
         bal.ext llink,sprintf      generate the message string        |
         space                                                         |
         la    r14,msgbuf           get address of message for strlen  |
         st    r14,0(r7)            and put it on c stack for strlen   |
         bal.ext llink,strlen       get the length of the message      |
         space
         lr    r13,r0               p$send needs length in r13     |
         bal.ext llink,p$send       send the message               |
         space                                                     |
quit     lis   r14,1                error exit status              |
         b.ext p$term                                              |
         space                                                     |
*        template for UNIX error message                           |
errmsg   db    c'reset: logical unit %d unassigned',x'0a',x'00'    |
msgbuf   ds    40                                                  |
         END



**P$READLN
P$READLN P.HEADR 01,00
         SPACE 5
* 7.6.8 PROCEDURE P$READLN(VAR T: TEXT);
*
* INTERFACE:
*        THE ADDRESS OF T'S FCB IS RECEIVED IN R14.
*
* RETURN:
*        IF EOF BECOMES TRUE, THEN THE CONDITION CODE ON RETURN IS
*        ZERO. OTHERWISE, IT IS FORCED NON-ZERO.
*        THIS FACT IS RTL-CONFIDENTIAL.
*
* OS32 ACTION:                                                     |
*        1. MAKE SURE THE FILE IS READABLE (I.E., RESET) AND
*           NOT AT EOF. IF NOT A MESSAGE IS LOGGED AND THE TASK
*           IS TERMINATED.
*        2. GET THE NEXT PHYSICAL LINE:
*           2.1 READ A PHYSICAL RECORD.
*           2.2 IF END-OF-FILE IS RETURNED, THEN SET EOF, FORCE THE
*               CONDITION CODE TO ZERO AND RETURN.  FOR OTHER I/O
*               ERRORS, LOG A MESSAGE, PAUSE, AND RETRY THE READ.
*           2.3 IF THE NUMBER OF BYTES TRANSFERRED IS < 256, THEN
*               PLACE A CARRIAGE RETURN CHARACTER AS A SENTINEL AFTER
*               THE LAST CHARACTER READ.
*           2.4 RESET THE TEXT BUFFER POINTER AND EOLN; INCREMENT THE
*               CURRENT FILE SIZE COUNTER.
*               IF THE CURRENT CHARACTER IS EOLCHAR, THEN SET
*               EOLN AND MAKE THE CURRENT CHARACTER BLANK.
*
* UNIX ACTION:                                                     |
*        1. SAME.                                                  |
*        2. GET THE NEXT LOGICAL LINE -- UNLIKE OS32, THIS REQUIRES |
*           MOVING THE FILE POINTER TO THE FIRST CHARACTER PAST THE |
*           NEXT NEWLINE CHAR IN THE FILE BUFFER.  THIS MAY OR MAY  |
*           NOT REQUIRE READING IN A NEW BUFFER FROM THE FILE.      |
*           ALGORITHM:                                              |
procedure p$readln (var t: textfcb);
  type textfcb: record {not exactly right, but you get
                            the point}
             eofflag, eolnflag: boolean;
             filesize: integer;
             bufr: array [1..textsize] of char
           end;
  var samebuf, search: boolean;
      tptr: integer;
  begin
    samebuf := true;
    search := true;
    tptr := 0;
    if not t.eolnflag then
      while search do begin
        tptr := tptr +1;
        if t.bufr[tptr] = chr(10) {newline char}
            then search := false;
        if tptr = textsize
            then begin
              readbuf (t);
              tptr := tptr - 1
            end
      end {while loop}
    else begin   {eoln true when entered routine}
      search := false;
      if (t.filesize = 0) or (tptr = textsize)
        then begin
            readbuf (t);
            samebuf := false
        end
    end; {else}
    if samebuf then tptr := tptr +1;
    if t.bufr[tptr] = chr(10)
      then begin
        t.bufr[tptr] := ' ';
        t.eolnflag := true
      end
  end; {p$readln}

procedure readbuf (var t: textfcb);
  var read_error: boolean;
  begin
    read_error := false;
    magicread (t); {gets a buffer, sets eofflag and
                    read_error}
    if read_error
      then begin
        print_error_message;
        terminate_process
      end;
    if not t.eofflag
      then begin
```

```
*                  t.filesize := t.filesize +1;                          |
*                  tptr := 0;                                            |
*                  t.eolnflag := false                                   |
*                end                                                     |
*            end; {readbuf}                                              |
*
* ERROR RESPONSE:
*        IF THE FILE IS NOT READABLE, OR EOF IS TRUE INITIALLY, THEN
*        LOG A MESSAGE AND ABORT.
*     OS32: FOR GENERAL SVC 1 ERRORS, CALL P$$SVC1 AND P$PAUS.           |
*     UNIX: FOR UNIX-DETECTED READ ERRORS, "PERROR" PRINTS THE MESSAGE.  |
*           THE PROCESS IS THEN TERMINATED.                              |
*
* CALLED FROM:
*        COMPILER GENERATED USER CODE.
*        P$GETT
*
* CALLS TO:
*     OS32: P$GETER1, P$GETER2, P$$SVC1, P$PAUS                          |
*     UNIX: P$GETER1, P$GETER2, read, perror, P$TERM                     |
         SPACE 5
         $PREGS LIST=NO
         $PASFCB LIST=NO
         TITLE P$READLN - GET NEXT INPUT RECORD (TEXT)
         P.DATA
         P.SAVEM R12
         ENDS
* definitions needed for UNIX                                            |
         udl                                                             |
         gbdata                                                          |
         options                                                         |
recflag  ds    2                  set = continue searching for newline   |
*                                 clear = quit search - found newline    |
         SPACE 2
P$READLN P.ENTER
         SPACE
* 1.
         L     R12,FCB.MW(R14)    GET FCB FLAGS
         TI    R12,MW.RESET       IS IT READABLE ?
         B.EXT P$GETER1,CC=Z      NO, FATAL ERROR
         TI    R12,MW.EOF         IS IT AT EOF ?
         B.EXT P$GETER2,CC=NZ     IF SO, TOO BAD
         SPACE
* 2.1
READLN.1 EQU   *
         ifnz  unix                                                      |
         lis   r13,1                                                     |
         sth   r13,recflag        set flag to continue getting chars     |
*                                      from bufr                         |
         l     r13,fcb.tptr(r14)  get ptr to current character           |
         l     r12,fcb.mw(r14)    get fcb flags                          |
         ti    r12,mw.eoln        at end of line?                        |
         bz    rdln.1a            no -- continue looking for newline     |
         space
         lis   r15,0
         sth   r15,recflag          clear flag to stop searching buffer
         ni    r12,-1-mw.eoln       reset eoln to false
         st    r12,fcb.mw(r14)
         l     r12,fcb.cfsz(r14)    is this the first record read?
         bz    rdln.1c              yes -- go read first record
         space
         l     r15,fcb.svc1+svc1.lxf(r14)   byte count from last read
         ai    r15,fcb.bufr-1(r14)   to find end of data
         clr   r13,r15              at end of data?
         bnl   rdln.1c              yes -- read a new record
         ais   r13,1                bump tptr to beginning of new line
         b     read.4a              and prepare to return
         space
* search for the newline char
rdln.1a  ais   r13,1                bump tptr to next char
continu  lb    r12,0(r13)           get the char
         cli   r12,newline          is it a newline char?
         bne   rdln.1b              no -- continue
         lis   r11,0                yes -- found nl - clear recflag
         sth   r11,recflag
* need a new buffer?
rdln.1b  l     r15,fcb.svc1+svc1.lxf(r14)   byte count from last read
         ai    r15,fcb.bufr-1(r14)   to find end of data
         clr   r13,r15              at end of data?
         bnl   rdln.1c              yes -- read a new record
         lh    r15,recflag          no -- get the flag
         bnz   rdln.1a              set -- get the next char
         ais   r13,1                clear -- bump tptr
         b     read.4a                   and prepare to return
         space
* read in a new buffer from the file
rdln.1c  l     r7,c.sp              get c stack ptr
         stm   r0,32(r7)            save pascal regs
         lr    r11,r14              move fcb address
         l     r15,udl.ext          get top of gbdata
         si    r15,gbdata           and get to the beginning of it
         lb    r12,fcb.svc1+svc1.lu(r11)  get the lu
         sla   r12,3                get fdtab offset
         ar    r12,r15              and calculate actual offset
         l     r13,fdtab+fd(r12)    get the filedes
         la    r14,fcb.bufr(r11)    get addr of buffer to receive data
         li    r15,textsize         number of bytes to be read
         stm   r13,0(r7)            parms for unix routine
         bal.ext llink,read         go do the read
         st    r0,fcb.svc1+svc1.lxf(r11)  save returned byte count
         lr    r0,r0                test return status
         bm    rderr1               read error
         space
         lm    r0,32(r7)            restore pascal regs
         l     r15,fcb.svc1+svc1.lxf(r14)  test byte count for eof
         bz    readln.2             go set eof flag
         lis   r11,1                                                      |
         am    r11,fcb.cfsz(r14)     increment current file size          |
         b     readln.4              continue                             |
         else  unix                                                       |
         SVC   1,FCB.SVC1(R14)       READ A RECORD
         LH    R15,FCB.SVC1+SVC1.STA(R14) CHECK STATUS
         BZ    READLN.3              OK
         SPACE
* 2.2
         TI    R15,S1ST.EMM+S1ST.EFM  EOF/EOM SET ?
         BNZ   READLN.2              YES, GO SET FLAG
         BAL.EXT LLINK,P$$SVC1       GO LOG ERROR MESSAGE
         BAL.EXT LLINK,P$PAUS        PAUSE FOR INTERVENTION
         B     READLN.1              THEN RETRY THE READ
         endc                                                             |
         SPACE
READLN.2 EQU   *
         l     r12,fcb.mw(r14)       restore fcb flags                    |
         OI    R12,MW.EOF            SET EOF FLAG
         ST    R12,FCB.MW(R14)       IN FCB
         LIS   R12,0                 FORCE COND CODE = ZERO
         B     READLN.5
         SPACE
* 2.3
READLN.3 EQU   *
         ifnz  os32                  UNIX does this automatically         |
         L     R13,FCB.SVC1+SVC1.LXF(R14) GET LENGTH OF XFER
         CLI   R13,TEXTSIZE          LESS THAN A BUFFER XFERED ?
         BNL   READLN.4              NO, IT'S OK AS IS
         LIS   R15,EOLCHAR           FORCE AN EOL AS SENTINEL
         STB   R15,FCB.BUFR(R13,R14) AT END OF BUFFER
         endc                                                             |
READLN.4 EQU   *
         SPACE
* 2.4
         LA    R13,FCB.BUFR(R14)     RESET TEXT POINTER
         ST    R13,FCB.TPTR(R14)     TO BUFFER START
         SPACE
         l     r12,fcb.mw(r14)       restore r12 with fcb flags           |
         NI    R12,-1-MW.EOLN        AND RESET EOLN
         ST    R12,FCB.MW(R14)
         ifnz  unix                                                       |
         lh    r15,recflag           get newline search flag              |
         bnz   continu               if set, continue                     |
         endc                                                             |
         SPACE
read.4a  LB    R12,0(R13)            CURRENT CHARACTER                    |
         ifnz  unix                                                       |
         ci    r12,newline           UNIX end-of-line?                    |
         else  unix                                                       |
         CI    R12,EOLCHAR           OS32 END OF LINE?
         endc                                                             |
         BNE   READ.5A               IF SO THEN...
         LI      R12,C' '             STORE A BLANK
         STB     R12,0(R13)           AND
         SPACE
         L       R12,FCB.MW(R14)      SET EOLN
         OI      R12,MW.EOLN
         ST      R12,FCB.MW(R14)
         SPACE
READ.5A  EQU     *
         SPACE
         ifnz    os32                 this done for UNIX earlier           |
         LIS     R12,1
         AM      R12,FCB.CFSZ(R14)    INCREMENT CURRENT FILE SIZE
         endc                                                              |
*                                     AND FORCE COND CODE NON-ZERO.
         SPACE
READLN.5 EQU     *
         st      r13,fcb.tptr(r14)    save new tptr                        |
         P.LEAVE                      AND RETURN
* UNIX-detected error in reading the file                                  |
rderr1   equ     *                                                         |
         l       r14,fdtab+fname(r12)  get the filename ptr                |
         st      r14,0(r7)            put it on c stack for perror         |
         bal.ext llink,perror         write the message                    |
         lis     r14,1                error return status for p$term       |
         b.ext P$TERM                 exit w/o return                      |
         END




**P$WRITLN
P$WRITLN P.HEADR 01,00
         SPACE 5
* 7.7.10 PROCEDURE P$WRITLN(VAR T: TEXT);
*
* INTERFACE:
*        THE ADDRESS OF T'S FCB IS RECEIVED IN R!4.
*
* OS/32 ACTION:                                                            |
*        1. MAKE SURE THAT T IS WRITABLE.
*        2. IF THE CURRENT TEXT POINTER IS LESS THAN THE SVC 1 BUFFER
*           END ADDRESS, THEN FORCE AN EOLCHAR AT THE END OF THE
*           BUFFER.
*        3. WRITE THE PHYSICAL RECORD.
*        4. RESET THE CURRENT TEXT POINTER TO THE BUFFER START ADDRESS
*           AND INCREMENT THE CURRENT FILE SIZE.
*
* UNIX ACTION:                                                             |
*        1. SAME                                                           |
*        2. IF THE CURRENT TEXT POINTER IS LESS THAN THE SVC 1 BUFFER      |
*           END ADDRESS, THEN FORCE A NEWLINE CHAR AT THE END OF THE       |
*           TEXT.                                                          |
*        3. WRITE THE BUFFER -- ONLY THE NUMBER OF CHARACTERS UP TO THE  |
*           CURRENT TEXT POINTER ARE WRITTEN.                            |
*        4. SAME                                                         |
*
* ERROR RESPONSE:
*   OS/32: IF AN SVC 1 ERROR OCCURS, CALL P$$SVC1 TO LOG AN ERROR        |
*        MESSAGE. THEN CALL P$PAUS TO PAUSE THE TASK.  UPON
*        CONTINUATION, RETRY THE WRITE.
*   UNIX: IF THE FILE HAS NOT BEEN REWRITTEN (NOT WRITEABLE), THEN       |
*        LOG THE ERROR MESSAGE WITH P$PUTERR.  IF AN ERROR OCCURS IN     |
*        THE UNIX "WRITE", THE UNIX-DETECTED MESSAGE IS PRINTED WITH     |
*        "PERROR." IF FEWER BYTES ARE ACTUALLY WRITTEN THAN WHAT IS      |
*        SENT TO "WRITE," THEN WRITE AN ERROR MESSAGE WITH P$SEND.       |
*        IN ALL CASES, THE PROCESS IS TERMINATED.                        |
*
* CALLED FROM:
*        COMPILER GENERATED USER CODE.
*        P$PUTT
*
* CALLS TO:
*    OS/32: P$PUTERR, P$$SVC1, P$PAUS                                    |
*    UNIX:  P$PUTERR, write, perror, sprintf, strlen, P$SEND, P$TERM     |
         SPACE 5
         $PREGS LIST=NO
         $PASFCB LIST=NO
         TITLE P$WRITLN - WRITE A PHYSICAL LINE TO A TEXT FILE
         P.DATA
         P.SAVEM R12
         ENDS
* definitions needed for UNIX                                            |
         gbdata                                                          |
         udl                                                             |
         options                                                         |
writsiz  ds    4                                                         |
         SPACE 2
P$WRITLN P.ENTER
         SPACE
         L     R12,FCB.MW(R14)      GET FLAGS
         TI    R12,MW.REWRT         AND ENSURE FILE IS WRITEABLE
         B.EXT P$PUTERR,CC=Z        IF NOT, FATAL ERROR
         SPACE
         L     R13,FCB.TPTR(R14)    GET TEXT POINTER
         ifnz  os32                                                      |
         SIS   R13,1                AND BACK IT UP 1
         endc                                                            |
         CL    R13,FCB.SVC1+SVC1.EAD(R14) BUFFER FULL ?
         BNL   WRITLN.1             YES, JUST WRITE IT
         ifnz  unix                                                      |
         li    r15,newline          else append newline char             |
         stb   r15,0(r13)                                                |
         else  unix                                                      |
         LI    R15,EOLCHAR          ELSE APPEND AN EOLCHAR
         STB   R15,1(R13)
         endc                                                           |
         SPACE
WRITLN.1 EQU   *
         ifnz  unix                                                     |
         l     r7,c.sp             get c stack ptr                      |
         stm   r0,32(r7)           save pascal regs                     |
         space                                                          |
         lr    r11,r14             save fcb address                     |
         lb    r12,fcb.svc1+svc1.lu(r11)  get lu for file descriptor    |
         sla   r12,3               calculate fdtab offset               |
         l     r14,udl.ext         get top of gbdata                    |
         si    r14,gbdata          then get to the beginning            |
         ar    r12,r14             and calculate actual offset          |
         space                                                          |
         l     r13,fdtab+fd(r12)   get fd                               |
         l     r14,fcb.svc1+svc1.sad(r11)  get buffer start address     |
         l     r15,fcb.tptr(r11)   for byte count                       |
         ais   r15,1                                                    |
         sr    r15,r14             calculate number xfer count          |
         st    r15,writsiz         and save it                          |
         stm   r13,0(r7)           put UNIX parms on c stack            |
         bal.ext llink,write       write the buffer                     |
         space                                                          |
         st    r0,fcb.svc1+svc1.lxf(r11)  save the transfer count       |
         lr    r0,r0               check the status (byte count)        |
         bm    writerr1            write error                          |
         cl    r0,writsiz                                               |
         bl    writerr2            wrote fewer bytes than sent          |
         space                                                          |
         lm    r0,32(r7)           restore the pascal regs              |
         else  unix                                                     |
         SVC   1,FCB.SVC1(R14)     WRITE THE LINE
         LH    R15,FCB.SVC1+SVC1.STA(R14) CHECK THE STATUS
         BZ    WRITLN.2            IF ZERO, CONTINUE
         BAL.EXT LLINK,P$$SVC1     LOG THE ERROR
         BAL.EXT LLINK,P$PAUS      AND PAUSE
         B     WRITLN.1            ON CONTINUATION, RETRY
         endc                                                           |
         SPACE
WRITLN.2 EQU   *                   WRITE SUCCESSFUL...
         LA    R13,FCB.BUFR(R14)   RESET TEXT POINTER
         ST    R13,FCB.TPTR(R14)   TO START OF BUFFER
         SPACE
         LIS   R13,1
         AM    R13,FCB.CFSZ(R14)   AND INCREMENT FILE SIZE
         SPACE
         P.LEAVE                   EXIT
* problem writing to file (UNIX write returned a -1)                    |
writerr1 equ   *                                                        |
         l     r15,fdtab+fname(r12) addr of filename ptr                |
         st    r15,0(r7)           put on c stack for perror            |
         bal.ext llink,perror      write the message                    |
         b     bye                 and exit                             |
        space
* wrote fewer bytes than sent to UNIX write
writerr2 equ    *
        la      r13,msgbuf           addr of buffer for message
        la      r14,ermsg            format string for sprintf
        l       r15,fdtab+fname(r12) get fname ptr for message
        stm     r13,0(r7)            put parms on c stack
        bal.ext llink,sprintf        create the message
        st      r13,0(r7)            put buffer ptr back on c stack
        bal.ext llink,strlen         get length of message
        lr      r14,r13              addr of message
        lr      r13,r0               length for P$SEND
        bal.ext llink,P$SEND         send it
bye     lis     r14,1                exit error status
        b.ext P$TERM                 exit with no return
*       format string for UNIX error message
ermsg   db      c'%s: wrote fewer bytes than sent',x'0a',x'00'
msgbuf  ds      80
        END



**P$NUMERR
P$NUMERR P.HEADR 01,00
        SPACE 5
* 7.6.10 PROCEDURE P$NUMERR(VAR T: TEXT);
*
* INTERFACE:
*       THE ADDRESS OF THE FILE'S FCB IS RECEIVED IN R14.
*
* OS32 ACTION:
*       AN ERROR MESSAGE IS CREATED AND LOGGED TO THE CONSOLE AND
*       THIS ROUTINE RETURNS TO THE CALLER TO HANDLE TERMINATION.
*
* UNIX ACTION:
*       AN ERROR MESSAGE WITH THE FILENAME IS CREATED AND SENT TO
*       STDERR. CONTROL IS RETURNED TO THE CALLER FOR TERMINATION.
*
* CALLED FROM:
*       P$$RDINT, P$READSR, P$READR
*
* CALLS TO:
*    OS32: P$SEND
*    UNIX: sprintf, strlen, P$SEND
        SPACE 5
        $PREGS LIST=NO
        $PASFCB LIST=NO
        TITLE P$NUMERR - INVALID CHARACTER IN NUMERIC INPUT
        P.DATA
        P.SAVEM R12
        ALIGN ADC
```

```
UNPKLU   DS    8
MESG     DS    132
         ALIGN ADC
         ENDS
* UNIX definitions                                                   |
mesg2    ds    80                                                    |
lmesg2   ds    2                                                     |
         options                                                     |
         gbdata                                                      |
         udl                                                         |
         SPACE 2
         align 2                                                     |
P$NUMERR P.ENTER
         SPACE
         ifnz  unix                                                  |
         l     r7,c.sp            get c stack ptr                    |
         stm   r0,32(r7)          save pascal regs                   |
         space                                                       |
         lb    r13,fcb.svc1+svc1.lu(r14)  get the lu                 |
         sla   r13,3              and shift for offset in fdtab      |
         l     r14,udl.ext        get to top of gbdata               |
         si    r14,gbdata         and then get to the beginning      |
         ar    r14,r13            calculate actual offset into fdtab |
         l     r15,fdtab+fname(r14)  get ptr to the filename         |
         la    r13,mesg2          address of message buffer          |
         la    r14,errmsg2        format string for sprintf          |
         stm   r13,0(r7)          put them on c stack for sprintf    |
         bal.ext llink,sprintf    create the message                 |
         space                                                       |
         la    r15,mesg2          address of message                 |
         st    r15,0(r7)          put it on c stack for strlen       |
         bal.ext llink,strlen     get the length of the message      |
         sth   r0,lmesg2          and save it                        |
         space                                                       |
         lm    r0,32(r7)          restore pascal regs                |
         else  unix                                                  |
         LA    R12,ERRMSG         MESSAGE ADDRESS
         LA    R13,MESG(R1)       ADDR OF DEST
         SVC   2,MOVMESG          MOVE IT TO THE STACK
         SPACE
         LHI   R12,X'C306'        UNPK DECIMAL, NLZ
         STH   R12,UNPKLU(R1)     FORM PARAMETER BLOCK ON STACK
         LA    R12,DEST.LU(R1)
         ST    R12,UNPKLU+4(R1)   SET DEST'N ADDR
         SPACE
         LR    R12,R0             SAVE R0
         LB    R0,FCB.SVC1+SVC1.LU(R14) GET BAD LU
         SVC   2,UNPKLU(R1)       UNPACK IT TO MESG
         LR    R0,R12             RESTORE R0
         SPACE
         LA    R14,MESG(R1)       ADDR OF MESG
         LI    R13,LMESG          LENGTH OF MESG
         endc                                                        |
         ifnz  unix                                                    |
         la    r14,mesg2             addr of UNIX mesg                 |
         lh    r13,lmesg2            length of UNIX mesg               |
         endc                                                          |
         BAL.EXT LLINK,P$SEND        LOG THE MESSAGE
         SPACE
         P.LEAVE                     RETURN TO CALLER
         SPACE 3
         ALIGN ADC
MOVMESG  DB    LMESG,18,R12,R13
         SPACE 2
ERRMSG   DB    C'INVALID CHARACTER IN NUMERIC INPUT, LU= '
LU       DB    C'XXX'
LMESG    EQU   *-ERRMSG
DEST.LU  EQU   LU-ERRMSG+MESG
*        template for UNIX error message                               |
errmsg2  db    c'%s: invalid character in numeric input',x'0a',x'00'   |
         END
```

Appendix B. The makefile used by the UNIX utility `make' for building the PEPascal compile-time and run-time environment. The development system can be built from scratch by issuing the command `make' while in the same directory where this file resides. Tests of individual rtl routines were made with `make test' where the CAL file "test.s" contains the test code. The PEPascal version available to the general public can be made with `make public'.

```
#The commented routines are not used in the UNIX port
#RTLOBJ1 = rtl/error/pas.rel.o rtl/fortran/p_fort.o

RTLOBJ1 = rtl/heap/p__remv.o rtl/heap/p_disp.o \
        rtl/heap/p_mark.o rtl/heap/p_new.o \
        rtl/heap/p_rel.o rtl/heap/p_spac.o \
        rtl/init/p_ermes.o rtl/init/p_init.o \
        rtl/error/pas.err.o rtl/init/decl.o

#RTLOBJ2 =
RTLOBJ2 = rtl/inputnt/p_read.o rtl/inputnt/p_get.o  \
        rtl/inputnt/p_reset.o rtl/inputt/p__rdint.o \
        rtl/inputt/p_gett.o rtl/inputt/p_readby.o \
        rtl/inputt/p_readch.o rtl/inputt/p_readi.o \
        rtl/inputt/p_readln.o rtl/inputt/p_readsr.o \
        rtl/inputt/p_readr.o rtl/inputt/p_readsi.o \
        rtl/inputt/p_resett.o rtl/inputt/dotatod.o \
        rtl/inputt/dotatof.o
#
RTLOBJ3 = rtl/iocommon/p__rewd.o rtl/iocommon/p_close.o \
        rtl/iocommon/p_efcb.o rtl/iocommon/p_ifcb.o \
        rtl/iocommon/p_rewrit.o  rtl/ioerror/p_geter1.o \
        rtl/ioerror/p_geter2.o rtl/ioerror/p_numerr.o  \
        rtl/ioerror/p_puterr.o rtl/ioerror/p_fcberr.o \
        rtl/ioerror/p__svc1.o rtl/ioerror/p__svc7.o

RTLOBJ4 = rtl/outputnt/p_put.o rtl/outputnt/p_write.o \
        rtl/outputt/p_page.o rtl/outputt/p_purge.o \
        rtl/outputt/p_putt.o rtl/outputt/p_writb.o \
        rtl/outputt/p_writch.o rtl/outputt/p_writln.o \
        rtl/outputt/p_writr.o rtl/outputt/p_writs.o \
        rtl/outputt/p_writsr.o rtl/outputt/pwrt.int.o \
        rtl/outputt/dotftoa.o rtl/outputt/dotdtoa.o

RTLOBJ5 = rtl/set/p_sand.o rtl/set/p_scomp.o \
        rtl/set/p_sdif.o rtl/set/p_sor.o \
        rtl/struct/p_filcpy.o rtl/struct/p_stcmp0.o \
        rtl/struct/p_stcmp1.o rtl/struct/p_stcmp2.o \
        rtl/struct/p_stcmp3.o rtl/struct/p_stcpy.o

#RTLOBJ6 = rtl/prefix/rename.o rtl/prefix/reprotec.o \
#       rtl/prefix/rewind.o rtl/prefix/back_fil.o \
#       rtl/prefix/back_re.o rtl/prefix/checkpoi.o \
#       rtl/prefix/forwd_re.o rtl/prefix/allocate.o \
#       rtl/prefix/delete.o
RTLOBJ6 = rtl/prefix/open.o rtl/prefix/close.o \
        rtl/prefix/fetch_at.o rtl/prefix/write_fi.o \
        rtl/prefix/forwd_fi.o rtl/prefix/breakpoi.o \
        rtl/prefix/start_pa.o rtl/prefix/time.o \
        rtl/prefix/date.o rtl/prefix/exit.o \
        rtl/prefix/p_iofun.o rtl/prefix/change_a.o \
        rtl/svcs/svc7.o rtl/prefix/prefix.o
RTLOBJA = $(RTLOBJ1) $(RTLOBJ2) $(RTLOBJ3)

RTLOBJB = $(RTLOBJ4) $(RTLOBJ5)  $(RTLOBJ6)

CRTLOBJ = comprtl/allotemp.o comprtl/dotatod.o \
        comprtl/readscan.o comprtl/setlink.o comprtl/readscn.o

#PASSES = ./passes/pass*.o
#PASSES = ./pas/passes.o ./passes/pass1.o ./passes/pass2.o \
        ./passes/pass3.o ./passes/pass4.o ./passes/pass5.o \
        ./passes/pass6.o ./passes/pass7.o ./passes/pass8.o \
        ./passes/pass9.o ./passes/pass10.o
PASSESA = ./pas/passes.o ./pas/pasloadr.o
PASSESB = ./pas/passes.o ./pas/pasldrall.o

DRIVER = ./pas/pascal.o

all:    pepascal

public: pasall

pepascal:  comprtl.o pasrtl1.o pasrtl2.o $(DRIVER) $(PASSESA)
        ld  -X -o pepascal /lib/crt0.o $(DRIVER) $(PASSESA)\
                pasrtl1.o pasrtl2.o comprtl.o -lc

pasall:    comprtl.o pasrtl1.o pasrtl2.o $(DRIVER) $(PASSESB)
        -ld  -X -o pasall /lib/crt0.o $(DRIVER) $(PASSESB)\
                pasrtl1.o pasrtl2.o comprtl.o -lc
        mv pasall /usr/pascal/pepascal
        chmod 775 /usr/pascal/pepascal
        touch pasall

# test used to test rtl routines with CAL programs
test:   test.o pasrtl1.o pasrtl2.o
        ld -X /lib/crt0.o test.o pasrtl1.o pasrtl2.o -lc

comprtl.o:      mutil.lib $(CRTLOBJ)
        -ld -r -x -o comprtl.o $(CRTLOBJ)

pasrtl1.o:       mutil.lib $(RTLOBJA)
        -ld -r -x -o pasrtl1.o $(RTLOBJA)

pasrtl2.o:       mutil.lib $(RTLOBJB)
        -ld -r -x -o pasrtl2.o $(RTLOBJB)
        calmacro < $*.s \
                MLIBS=mutil.lib MLIST=NONE
        as -u -o $*.o m.out.s
        rm m.out.s

mutil.lib:      macros.src macros.ksu
        cat macros.ksu macros.src | bldlib
        rm mutil.tmp


rtl/prefix/prefix.o:    rtl/prefix/prefix.c
        cc -c rtl/prefix/prefix.c
        mv prefix.o rtl/prefix

comprtl/readscn.o:      comprtl/readscn.c
        cc -c comprtl/readscn.c
        mv readscn.o comprtl
```

Appendix C. On-line UNIX manual entries for the Perkin-Elmer Pascal compiler and run-time environments. The former is obtained on-line with the command `man pascal' while the latter is invoked with `man 7 pascal'.

NAME

pascal - Perkin-Elmer Pascal compiler

SYNOPSIS

pascal name [options] [-o file]

DESCRIPTION

pascal compiles the Perkin-Elmer Pascal program in _name_. Note that the named file must have the .p extension but the .p may or may not be included in the file name on the command line. Normal compilation messages are sent to stderr unless the NLOG option is specified. If no compilation errors occur, the executable objectfile is placed in the file a.out in the current directory. Otherwise, compiler error messages are sent to stdout with the corresponding source line number. These error messages are imbedded in the source code listing if the LIST option is used. Entering the command name with no arguments will echo a synopsis of the use of this command.

A number of options are available. They may not be concatenated together (like -vt must appear as -v -t). Any option starting with a '-' and not listed below is passed on to the link editor, _ld_(1). Any option not starting with a '-' is used by the compiler. Most of these compiler options may also be included in the source code in a comment of the form {$x[+-]}, where 'x' is the compiler option mnemonic, '+' means turn on the option, and '-' means turn it off.

-o The _file_ argument after -o is used as the name of the _ld_ output file, instead of a.out.

-c compile-only. Compiler output is not loaded and is placed in _name_.o

-k_n_ To expand the compiler work space by _n_ bytes where _n_ is an integer value optionally followed by "k" to specify multiplication by 1024, "b" for 512, and "w" for 4. The default memory allocation for the compiler is 128k.

-t Uses the current directory as the library instead of /usr/pascal/lib.

-v Verbose. Although the Perkin-Elmer Pascal compiler is normally fairly informative, this option tells you even more, perhaps to the point of annoyment.

Compiler options (all output goes to stdout); to turn off the option, precede the option name with an 'N':

AS¦ASSEMBLY
: Prints an assembler listing of the object program. Default is NAS.

BO¦BOUNDSCHECK
: Checks for illegal values assigned to variables of subrange type. Default is BO.

CR¦CROSS
: Produces a cross reference listing of the program's identifiers. Default is NCR.

EJECT
: Produces a page eject in the compiled program's output listing. NOTE: this option can only be used in-stream.

HE¦HEAPMARK
: Causes the compiler to recognize the routines MARK and RELEASE as standard procedure identifiers. This option must be used for any program which has references to these procedures in it. Default is NHE.

LI¦LIST
: Prints a listing of the source program. Default is NLI.

LO¦LOG
: Prints to stderr notices of compiler operations, such as current pass number and the number of errors encountered (if any). Default is LO. NOTE: This option may only appear in the command line - it cannot be used in-stream.

MA¦MAP
: Prints the code displacements and data area displacements by line number. Default is NMA.

OP¦OPTIMIZE
: Causes compiler to perform optimizations. Default is NOP. NOTE: This option may only appear in the command line - it cannot be used in-stream.

SU¦SUMMARY
: Prints a summary of the optimizations performed, intermediate code size, and heap use. Default is NSU.

RA|RANGECHECK
Causes generation of code for run-time range checking of subscripts, case labels, variant tags, pointer values, and constant subrange parameters. Default is RA.

SEE ALSO
ld(1), a.out(5).
pascal(7) for a description of the run-time environment
Perkin-Elmer Corp., Pascal user guide, language reference, and run time support reference manual. Pub. no. 48-021R01, 1982.

BUGS
The compiler options BATCH, BEND, INCLUDE, and RELIANCE are not currently implemented. The MEMLIMIT option is implemented but only wastes space since it causes memory to be allocated which is inaccessible.
Very large programs may require more memory for the compiler. See the -k option above to get more memory for the compiler.

CREDITS TO
Dept. of Computer Science, Kansas State University, Manhattan

NAME

pascal - Perkin-Elmer Pascal run-time environment

SYNOPSIS

a.out [-kn] [-d] [-fn file] ... [program args]

DESCRIPTION

To run a Perkin-Elmer Pascal program, execute the compiled and linked object file produced by _pascal_(1). This file is _a.out_ by default, or the file named by the -o option for _pascal_(1), if used. The options are:

-k_n_ To expand the user work space by _n_ bytes where _n_ is an integer value optionally followed by "k" to specify multiplication by 1024, "b" for 512, and "w" for 4. The default work space size is 8k.

-d Produces a dump of the core image in the file "core" in the current directory in response to certain run-time errors for debugging. Default is to not dump the core.

-f_n_ file
For interaction with files. The _file_ argument is the UNIX pathname associated with the logical file indicated by _n_. The logical file number corresponds to the position of the file identifier in the program header file-name list. The first file in the list has a logical number of 0, the second file is number 1, the third number 3, and so on. There is a limit of 32 external files, although UNIX limits a process to 20 open files. Since stdin, stdout, and stderr are always open for a process, you are limited to 17 files (both internal and external) open at a time. To make the program interactive, you can map your input and output files to /dev/ttyx where x is the tty number of the terminal where you are working. For example, the program header "program main (input, output);" would require the following command line at run-time to make it interactive:

```
a.out -f0 /dev/ttyx -f1 /dev/ttyx
```

Any other arguments are passed on to the program. These arguments can be accessed from within the program with the START_PARMS prefix routine.

SEE ALSO

pascal(1), a.out(5).
Perkin-Elmer Corp., Pascal user guide, language reference, and run time support reference manual. Pub. no. 48-021R01,

1982.

BUGS

The external files INPUT and OUTPUT are treated like any other files at run-time. Future efforts will map these into stdin and stdout, respectively.

CREDITS TO

Dept. of Computer Science, Kansas State University, Manhattan

A UNIX PORT OF THE PERKIN-ELMER PASCAL RUN-TIME LIBRARY

by

HARVARD CHARLES TOWNSEND

B.S., Kansas State University, 1980

AN ABSTRACT OF A MASTER'S REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1984

ABSTRACT

This project involved porting Perkin-Elmer's Pascal run-time library (RTL) from the OS/32 operating system to the UNIX operating system for Perkin-Elmer 32-bit minicomputers. In Perkin-Elmer Pascal, the RTL acts as the interface between the user program and the underlying operating system (OS). Rtl routines may request OS services, such as file input/output or memory allocation, to help carry out a function requested by the compiled user program. The porting process therefore concentrated on translating OS/32 service requests to UNIX requests. For programmer efficiency, C programming language library routines were used to interface with the UNIX kernel. Four OS-dependent issues influenced the porting process: the interface with the underlying OS, memory management, error handling, and file handling. The differences in how the two operating systems treat these four areas led to specific implementation needs. These are discussed both conceptually and in terms of specific implementation details. This project took approximately 3 man-months to complete which is consistent with ports of other languages.