PERQ™ System Software

Reference Manual

February 1982 Software Version D.6

Textual Material Prepared by
Three Rivers Computer Corporation
Pittsburgh, Pennsylvania

TABLE OF CONTENTS

INSTALLATION GUIDE

1.1 Unpacking Instructions.

To unpack base unit (the big box), refer to Drawing #0410-A.

1. Open top of largest box.
2. Turn box upside down by carefully "rolling" it over.
3. Open inner box (the side which is now up). Note that the feet are up.
4. "Roll" box over and lift it off.

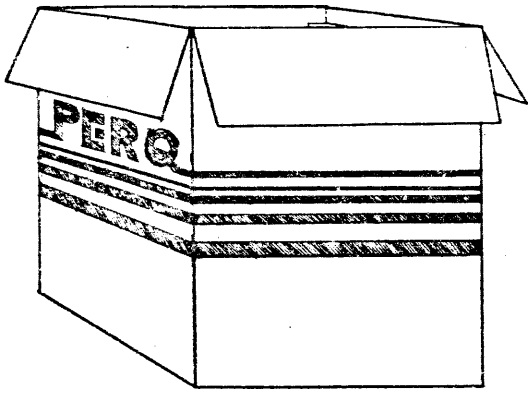
1.2 Unit Installation.

Be certain that the PERQ is NOT connected to a power source.

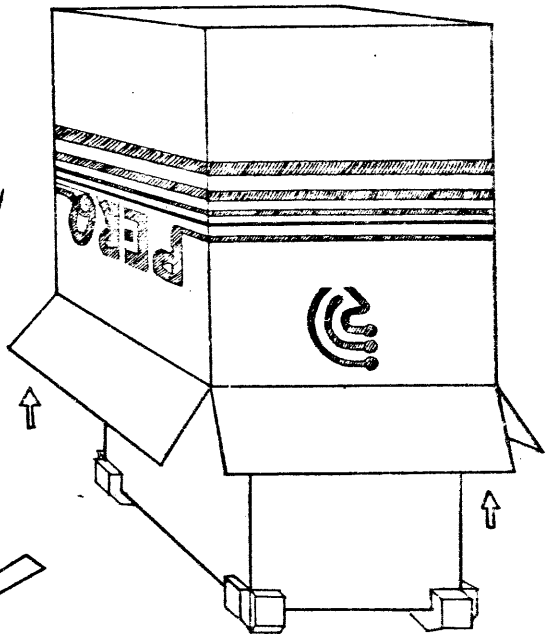
1. Position the PERQ in its desired location. Allow at least six inches between the PERQ and an obstruction such as a wall. The clearance is required to provide access to the screws securing the back or side covers.
2. Lower the four levelers 3/4 inch to permit airflow beneath the machine. Adjust the leverlers to compensate for uneven floors and make the PERQ stable.
3. Using a #2 Phillips screwdriver, remove the four (4) front cover screws. See drawing #0133-A. Remove the front cover.
4. Similarly, remove the rear cover. See drawing #0134-A.
5. When facing the front of the PERQ, the harddisk drive is located on the left hand side. Remove the two (2) screws securing the left side cover. Remove the left side cover. See drawing #0135-A.
6. Using the supplied Allen Wrench, remove disk shipping screw. See drawing #0137-B. Do not lose the shipping screw because it must be reinstalled before you move or ship the PERQ.
7. Remove shipping disk from floppy drive.
8. Plug in A.C. connector for disk. See Drawing #0137-B.

CAUTION - DURING THE FOLLOWING STEPS, HAZARDOUS VOLTAGES ARE PRESENT IN THE CHASSIS.

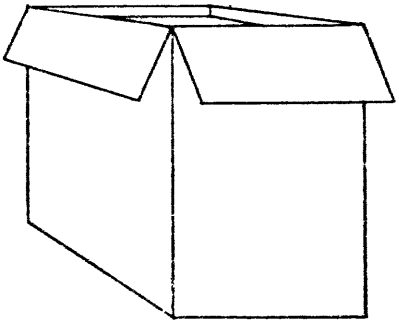
9. Plug line cord into appropriate power sources. See label on PERQ above line conncector for power requirements.



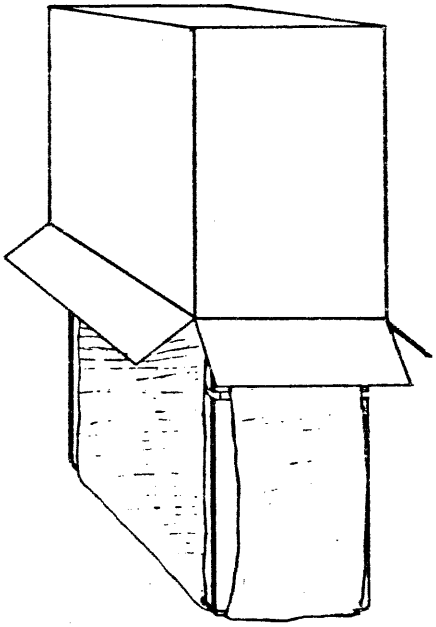
1



2



3



4

Three Rivers Computer	
PERQ UNPACKING ILLUSTRATION	PERQ
	PRQ-AD-0410-A

10. Pull front and rear "cheat" switches. Operate "On/Off" switch on front of PERQ. System will turn on. Wait 30 seconds.

11. Remove disk head locking clip ("A" Clip) by pulling in direction of arrow. See Drawing #0137-B. Do not lose this lock because it must be reinstalled before you move or ship the PERQ.

12. Operate "On/Off" switch to turn PERQ off.

13. Remove line cord from wall outlet and then from PERQ.

14. Re-install left side cover. (Note labels on cover.)

15. Re-install rear and front covers.

16. Plug in keyboard, display, and tablet (bit pad) to rear of PERQ. Connect the tablet cable between plug J7(GPIB), at the rear of the PERQ, and the back of the tablet. DO NOT USE PLUB J2 (tablet). Tighten the thumb screws on both plugs. Connect the power supply cable to the back of the tablet, then plug the power supply into the wall outlet.

17. Plug line cord into appropriate power source. See label on PERQ above line connector for power requirements.

NOTE: Shugart documentation regarding SA4000 (rigid disk) is enclosed for reference.

The Federal Communications Commission of the United States Government has published regulations which govern the allowable limits of emanation of radio frequency energy of computing devices and associated peripherals.

Those regulations are concerned with interference to radio communication, such as radio and TV.

The regulations require equipment for end use in the United States to be labeled and to be accompanied by the notice appearing in this instruction manual.

> Warning: This equipment generates, uses and can radiate radio frequency energy and if not installed and used in accordance with the instructions manual, may cause interference to radio communications. As temporarily permitted by regulation it has not been tested for compliance with the limits for Class A computing devices pursuant to Subpart J of Part 15 of FCC Rules, which are designed to provide reasonable protection against such interference. Operation of this equipment in a residential area is likely to cause interference in which case the user at his own expense will be required to take whatever measures may be required to correct the interference.

1.3 Reshipment Procedures

This section details the procedures to follow if you must move or

ship your PERQ. Sectins 1.3.1 and 1.3.2 describe the packing procedures.

1. Turn power off and disconnect all external sub-assemblies.

2. Using a #2 Phillips screwdriver, remove the four (4) front cover screws. See drawing #0133-A. Remove the front cover.

3. Similarly, remove the rear cover. See drawing #0134-A.

4. Remove the two (2) screws securing the left hand side cover (left side when facing the front of PERQ) and remove the left side cover.

5. Reconnect the monitor and keyboard, pull both "pull to cheat" switches, and power on the macine.

6. Boot the PERQ and Log in.

7. Type BYE WAIT. This moves the disk heads to the center of the disk.

8. Insert disk head locking clip ("A" clip). Push the "A" clip to install. Do NOT install the "A" clip unless the disk is spinning.

9. Turn power off.

10. Disconnect the monitor and keyboard. Disconnect the AC power plug from the hard disk (see drawing #0137-B).

11. When the disk has stopped spinning, turn the motor and align the hole for shipping screw. Insert shipping screw using Allen Wrench.

12. Replace left side cover. Replace front and rear covers.

1.3.1 Packing Procedures

1. Take the two longest pieces of white Micro-foam and lay them on the floor in the shape of a cross (+). Place the PERQ on top of the pieces and wrap the pieces over the PERQ.

2. Slide the box about the size of the PERQ over the foam wrapped machine.

3. Very carefully, turn the box over so that the open end of the box is up. Lower the four (4) levelers on the botoom of the PERQ. Tape the box shut.

4. Place the corner-shapes on the corners of the boxed PERQ.

5. Carefully, slide the large outer box over the boxed PERQ and turn the large outer box over so that the open end is up. Tape the outer box shut.

1.3.2 Monitor and Sub-assemblies Packing

1. Wrap the monitor with white Micro-foam. Find a box about the size of the monitor and slide it over the monitor. Tape the box shut.

2. Place white and green plastic corners on the monitor box. Place a larger outer box over the boxed monitor and tape shut.

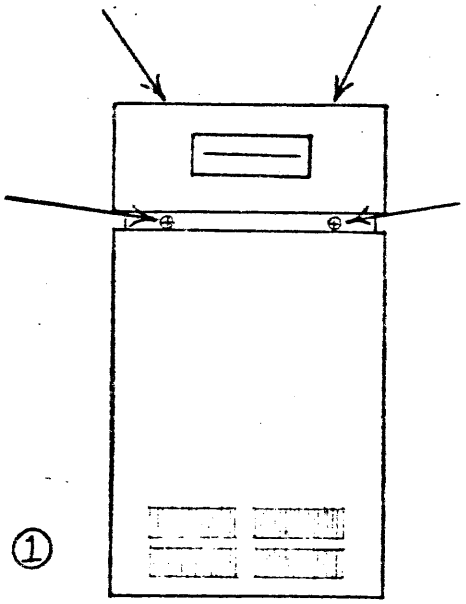
3. Find the box the size of the keyboard. Insert keyboard, close, and tape shut.

4. Insert the bit pad, bit pad magnet, and manual into the bit pad box and tape shut.

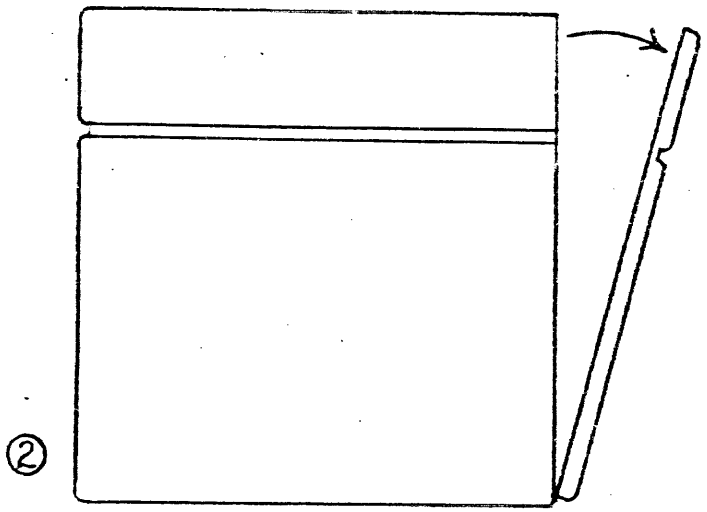
5. Box the cables and power cords and tape shut.

6. Take all boxes and place them in the larger outer box. Pack with foam to take up extra space. Close and tape.

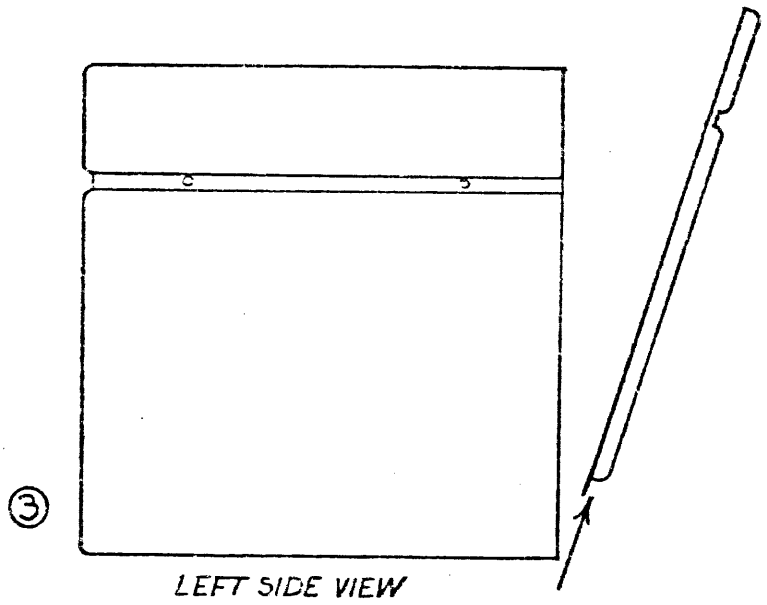
7. After packing and sealing the boxes, they should be palletized to eliminate shipping damage.



① FRONT VIEW

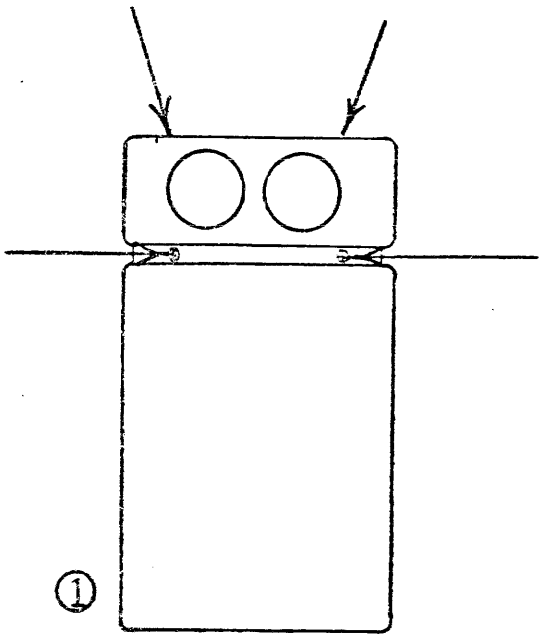


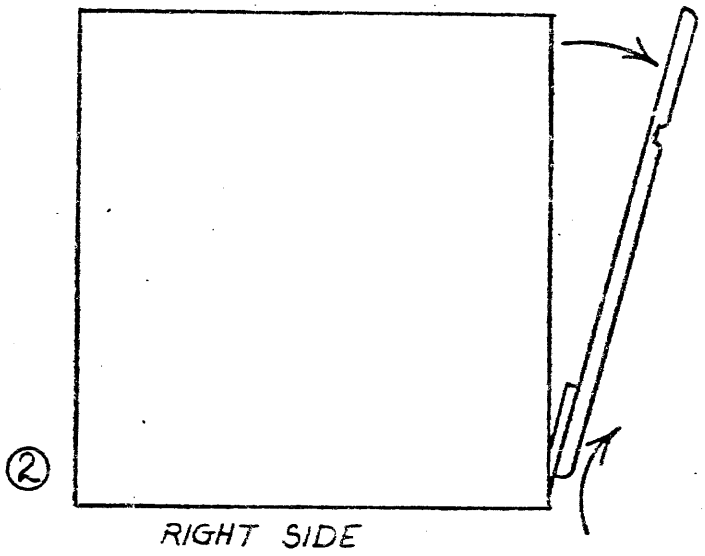
② LEFT SIDE VIEW

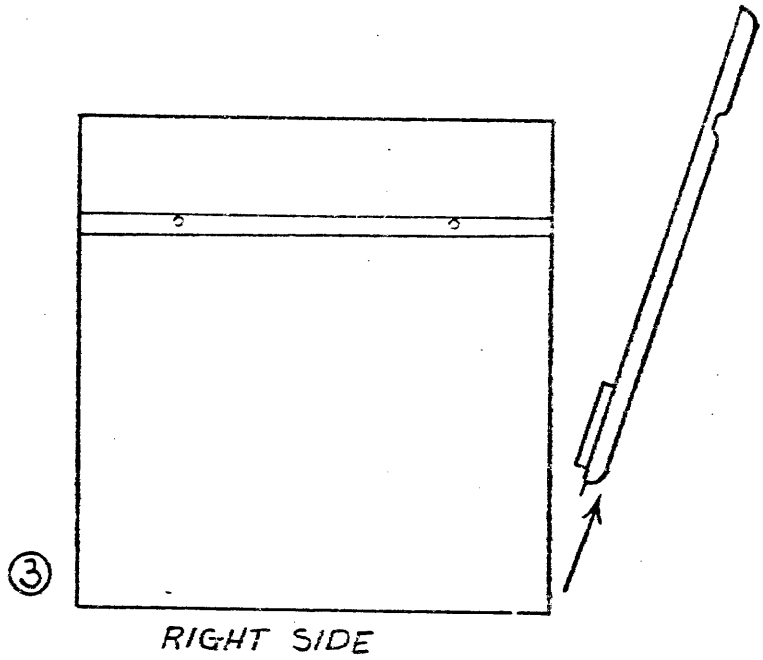


③ LEFT SIDE VIEW

PERQ BASE UNIT:
REMOVING THE FRONT COVER







PERQ BASE UNIT

REMOVAL OF REAR COVER

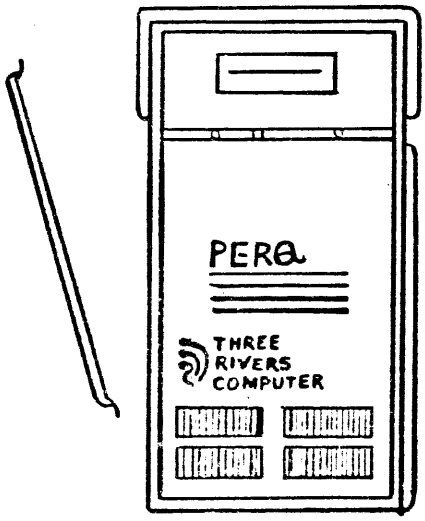
1 REMOVE SCREWS (SHOWN BY ARROWS)

2 TILT AND LIFT COVER SIMULTANEOUSLY

3 REMOVE COVER

REVISIONS

LTR	ECO NO.	DATE	APPV'D
B	00078	2-2-82	[illegible]



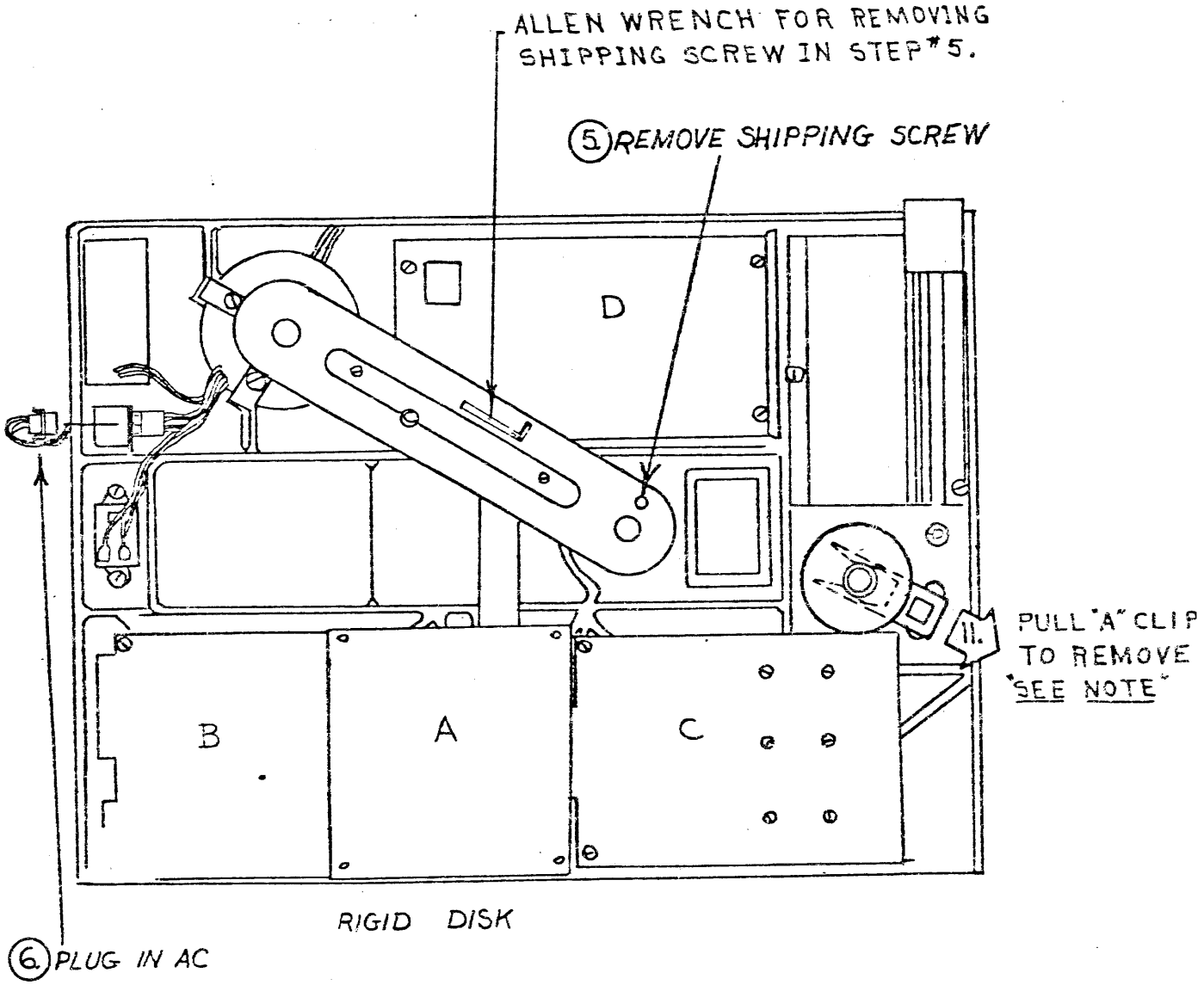
FRONT VIEW

NEXT ASSY PAK-001

TITLE LEFT SIDE COVER REMOVAL

Three Rivers Computer

			SIZE	CODE	IDENTIFICATION	VAR	REV
DRAWN	DIV	2-2-82	A	PRQ	AD-0135		B
CHECKED	[illegible]	2-8-82					
APPV'D	[illegible]	2-2-82	PROJ	PERQ		SHT	2 OF 4



NOTE: DO NOT REMOVE "A" CLIP, UNLESS PERQ IS "ON" AND DISK IS ROTATING. SEE INSTRUCTION 1.2.10 & 1.2.11

SA4000 UNPACKAGING INSTRUCTIONS

CAUTION: These directions **must** be carefully followed to insure the correct operation of the drive.

1. The **SPINDLE LOCKING SCREW must** be removed **before** applying AC power to the drive motor.

   IMPORTANT: Retain locking screw for re-installation prior to transporting drive.

2. **CAUTION: ROTATE SPINDLE IN DIRECTION OF ARROWS ONLY.**

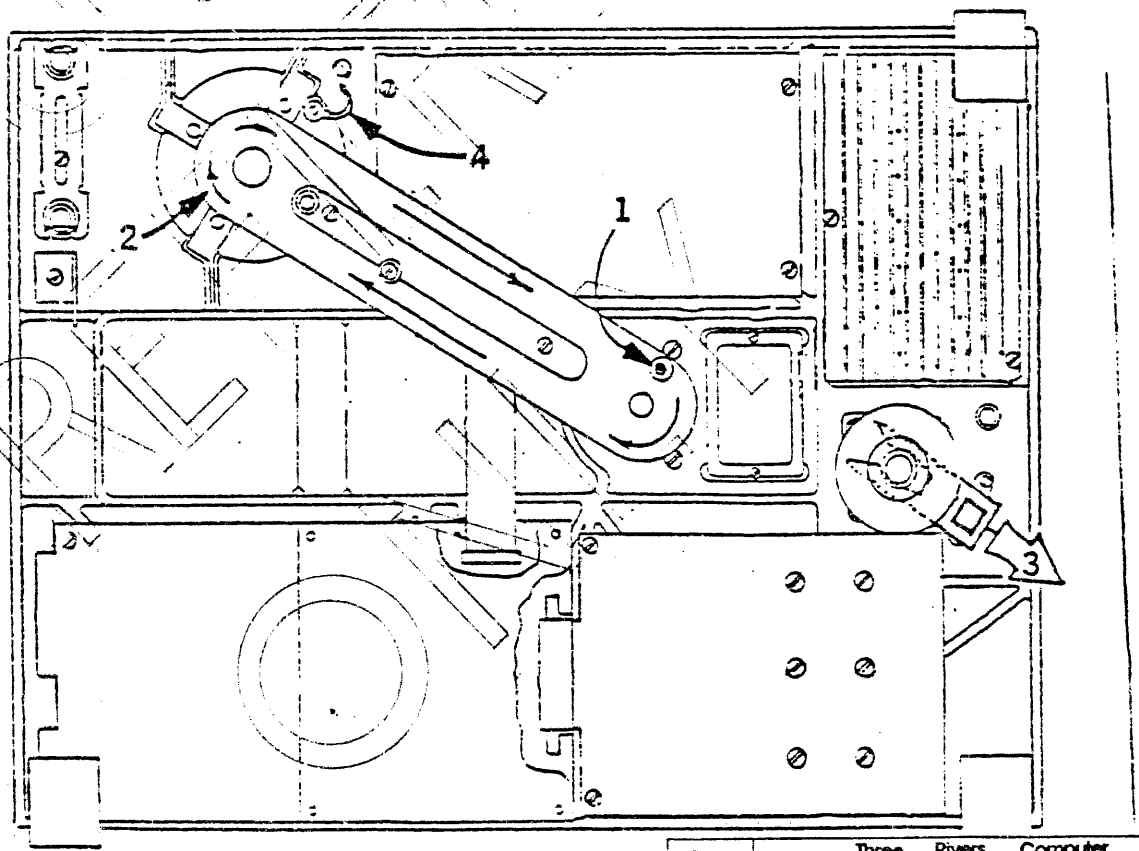
3. **Remove the ACTUATOR LOCK** in the following sequence:

   1. Energize AC power
   2. Withdraw the lock from the stepper assembly.

   IMPORTANT: Retain the lock for re-installation prior to transporting the drive.
   It is recommended the heads be moved to the extreme inside tracks prior to re-installing the actuator lock.

   CAUTION: It is recommended that the AC power be energized before removing or installing the actuator lock. Failure to do this may result in **media damage.**

4. OPTION — To isolate AC motor - remove GND strap.



Three Rivers Computer		
REFERENCE INST.	PROJECT PERQ	
DRAWN BY T.T.	DRAWING NO. PRQ-AD-0611-A	SHEET 1 of 1
CHECKED BY		

PERQ Introductory User Manual

Donald A. Scelza

Diana Connan Forgy

Brad A. Myers

Bob Amber

This manual provides an introduction to the use of the Three Rivers Computer Corporation PERQ Computer.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

1. Preface

This manual provides an introduction to the PERQ. The manual explains general concepts of the PERQ system; you should be familar with the concepts before using the PERQ. This manual also supplies information to boot the PERQ system.

2. Introduction.

PERQ is Three Rivers Computer Corporation's personal computer which provides an integrated computing system for a single user. All PERQs come with the following hardware features:

a) 16-bit micro programmed processor
b) High-resolution graphics using a 1024 X 768 bit mapped raster display
c) Standard keyboard with special function keys
d) 12-megabyte Winchester technology disk drive
e) RS232 & GPIB I/O interfaces
f) 512k bytes of main memory
g) Pointing tablet
h) Speech output

Optional features are:

a) 24 megabyte disk
b) 4k X 48-bit writable control store
c) 1 megabyte of main memory
d) 8 inch floppy disk drive

PERQ provides a large, 32-bit, segmented virtual address space. Virtual addresses are mapped into a 20-bit physical address.

The supplied software provides a functional program development environment. The system includes a text editor, a Pascal compiler, interactive stack dump for debugging, file management utilities, and support for micro program development.

3. Turning the PERQ on and off.

Refer to the Installation Guide before using your PERQ.

The PERQ's power switch is located on the front panel in the groove below the floppy disk drive. The switch is right of center and is labeled OFF/ON. Pushing the switch to the right powers on the PERQ. The fans start; if you do not hear them check to see that the machine is plugged in. When the PERQ is plugged in, the small neon light in the lower right hand side of the back glows. If the light is out, check the electrical outlet. If there is power at the outlet or the light is glowing and you still do not hear the fans then call your local service representative.

The PERQ takes about two minutes to boot after it is turned on allowing the disk to spin up to speed. At this point the Diagnostic Display (DDS) reads 999. Refer to Section 12 for boot procedures and information on the DDS.

If the machine has not booted after two or three minutes, try pressing the Boot button on the back of the keyboard. If this does not cause the machine to boot, call your local service representative.

After the PERQ boots, you will be asked to enter the time of day. The time and date is given in the form:

DD MMM YY HH:MM:SS

The time notation uses a 24-hour clock. If you were logging in on the 27th of March, 1982 at 2:30 in the afternoon you'd type

27 mar 82 14:30:00

The seconds are optional and the spaces in the date can be replaced by hyphens (27-Mar-82). There is usually a default date and time given at login. If the date is accurate, enter only the time or, type a carriage return to confirm both date and time. If it is not accurate or if there is no default, enter the correct date and time.

Next, you will be asked to login. You must supply a user name and password to be allowed access to the machine. After the operating system has been loaded onto a machine, the names defined are "Guest" and "" (the empty string; just press the Carriage Return key). See "UserControl" in the Utility Programs Manual for instructions on how to add other user names. After you have successfully logged in, the command interpreter, Shell, will be loaded and you will be able to execute commands.

Before turning the PERQ off, you should always log off using the Bye program. If you just power down, some temporary files on the disk will not be cleaned up. Type:

```
BYE OFF
```

to log off and turn the power off for the machine automatically. If this is not possible, you can always turn the machine off by moving the power switch to the left. See "BYE" in the Utility Programs Manual.

4. Control Characters and Special Keys.

The PERQ operating system recognizes a number of special control characters. These control characters perform simple input line editing and program control.

The valid control characters are:

BACK SPACE or ^H - erases the last character typed by the user.

^W or ^BACK SPACE - erases the last word typed by the user.

OOPS or ^U - erases the last line typed by the user.

^C - typed once causes a current program to abort the next time that it asks the operating system for input.

^C - typed twice causes the current program to abort immediately. However, this does not cause user command files to exit; see ^SHIFT C.

^S - causes program output to the screen to be suspended.

^Q - allows output to the screen to resume after a ^S.

^SHIFT C - causes a dump of the runtime stack and an immediate return to the Shell. If a command file was being executed, it is aborted and control is returned to the keyboard.

^SHIFT D - causes a dump of the runtime stack to be printed and the preliminary debugger (called Scrounge) can then be entered. If the debugger is not entered, the original program will resume execution. If the debugger is used, the user can request that the program be resumed after investigating the state. For a more complete description of the debugger, see the "PERQ Utility Programs Manual."

HELP - pressing the HELP key by itself displays a general help message. If you press the key after typing a command, the display contains specific help information for the command.

5. The Command Interpreter, The Shell.

The command interpreter for the PERQ operating system runs as a separate user program. It is possible for the user to replace the command interpreter with his own (see the section on "Login" in the PERQ Utility Programs Manual). The name of the command interpreter supplied with the system is the Shell.

The Shell takes commands from the user terminal or from a user command file and executes them. It does not distinguish between upper and lower case letters; you can use whichever you prefer. The commands in a user command file look exactly as if they were typed at the terminal. However, the cursor is a lighter shade so that you can distinguish command file execution from a typed command. The PERQ Utilities Programs Manual describes user command files and how to use them.

The general form of a Shell command line is a command or program name, followed by any number of optional parameters, followed by any number of optional switches. Some of the switches take parameters. For example:

```
Compile SourceProgram/Symbols=32
```

compiles a Pascal program in file SourceProgram using 32 symbol table blocks. All switches begin with a "/".

An example of a program that takes a number of parameters on the command line is the Copy program. The form is:

```
Copy SourceFile~DestinationFile
```

An example of a program that takes multiple switches is the Linker. A Linker command line might be:

```
Link SourceProgram/Verbose/StackSize=1024
```

When you supply a command line to the Shell, it extracts the first symbol on the line and does a unique substring lookup of the symbol against a small set of commonly used commands. You can get a list of these commands by typing "?" or the HELP key. If a match is found, the Shell executes the command. If no match is found, the Shell assumes the symbol is the name of an executable runfile (the output of the Linker), and attempts to execute it. Commands can also be invoked by means of a menu if you have booted from the harddisk. See the section on "PopUp Menus" below.

6. Specifying Commands and Arguments.

The Shell uses a default file name as the parameter to certain programs if no parameter is provided. This file name is the last file name typed to one of these programs. The Editor, Compiler, and Linker use and set the default file name. The TypeFile program uses the default file name but does not set it. The PERQ Utility Programs Manual provides more details on these programs.

Other programs require that you specify arguments. If a command requires an input and an output argument, you can specify the arguments as either

```
INPUT~OUTPUT
```

or

```
INPUT OUTPUT
```

You can separate the input and output arguments with multiple spaces; only the initial space delimiter is relevant.

However, if a command accepts multiple input or output arguments, you must separate like arguments with commas (,) and distinguish input from output with the tilda character (~). For example:

```
input1,input2,...inputn ~ output1.output2,...outputn
```

If a command accepts multiple input arguments and no output arguments, you must separate the arguments with a comma (,).

Switches modify the action of the command and therefore must follow the command specification. An exception is the Help switch (either type /HELP or press the HELP key); when specified before the command, the Help switch supplies general information and when specified after a command, the Help switch provides specific information. Switches always start with a slash (/) and are generally optional. If a switch accepts a parameter, specify the parameter after the switch, but preceded by an equal sign (=). For example:

```
/switch=parameter
```

The effect of a switch is global; regardless of where the switch appears on the command line, it has the same effect. A switch applies to every argument. If a command accepts multiple input or output arguments, no switch applies to one and not another argument. The PERQ Utilities Programs Manual provides details on the general syntax.

For most programs on the PERQ, arguments that have defaults will print the default answer in square brackets ("[]"). This answer can be chosen by simply typing a carriage return (thus providing an empty argument). To supply a different value, type the value followed by a carriage return. For example, if:

```
Delete FileName.Seg [No]:
```

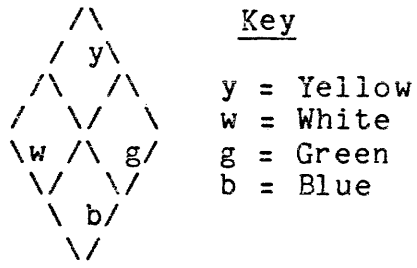
is the prompt for an argument, a carriage return means no. Of course, you could type "yes" or "no" as the argument. Most programs do not distinguish between upper and lower case for arguments or commands. In addition, you can abbreviate a command to the number of letters unique to other command names. However, this does not work for filenames.

The user's profile file specifies the set of commands recognized by the Shell. You can use a copy of the file DEFAULT.PROFILE initially. Later, you may wish to define commands of your own. Refer to the section on "Profiles" below.

7. The Pointing Device.

PERQs are supplied with a bit pad and pen or puck. In the normal mode, the cursor on the screen follows the movement of the pen. If the pen is in the upper-left corner of the tablet, the cursor will be in the upper-left corner of the screen. The PERQ can read the pen position when the pen is near the tablet surface. If you press down on the pen, a small switch closes. This is called a "press" of the pen. The editor and some other programs use the pen for pointing and drawing.

As an option, the pen can be exchanged for a four-button puck. The puck rests on the tablet and uses a circle with cross hairs for positioning. The four buttons are arranged in a diamond as shown below. In all cases, a press of the top (yellow) button on the puck functions exactly like the press on the pen. For programs which do not distinguish between the buttons, the other buttons also act like a press. The editor, however, assigns different functions to the other three buttons (see the "Editor User's Guide").



8. PopUp Menus.

The Shell and the FLOPPY and FTP utility programs allow their arguments to be entered by means of menus. (This only works, however, if you have booted from the harddisk.) The menu holds a list of all legal commands or arguments, and the pointing tablet can be used to specify the selection. Using a menu is sometimes more convenient than typing out the name of the command or argument desired. PopUp menus appear when requested. When the selection is made, the menu disappears and restores the screen space that was covered by the menu. Using PopUp menus, therefore, does not require sacrificing any screen area.

For the Shell, you can invoke a menu whenever the prompt (">:") is displayed and no characters have been typed. (The prompt is actually a dark grey, right pointing triangle.) To invoke a PopUp menu, simply press the pen or puck. The menu appears at the current cursor position. The cursor can then be moved up or down inside the menu to select the desired command. The selected command is highlighted by reverse video. A press over the selected command causes it to be invoked just as if the user had typed the command on the keyboard.

If the menu is not large enough to hold all the commands, a scrolling mechanism is provided. When scrolling is necessary, a "gauge" is displayed in a black border at the bottom left of the menu. When you move the cursor over this area, the cursor changes to a scroll. A press here allows scrolling. Moving the cursor to the right while pressing causes the menu text to scroll up and moving the cursor to the left while pressing causes the text to scroll down. The further the cursor is moved from the original press position, the faster the text scrolls. A line in the gauge shows how fast the menu is scrolling. Of course, you cannot scroll past either end of the menu. (Each end is signified by a row of "~"). Releasing the button causes scrolling to stop.

If you press in an illegal part of the menu, or if you try to invoke a menu and have typed some text, the PERQ beeps. If a menu is displayed and you press outside of the menu, the menu disappears, causing no side effects. In addition, if you type ^C or ^Shift-C while a menu is displayed, the menu disappears.

9. The "Lights".

The PERQ does not have a front panel full of lights like many computers. To show the occurrence of certain long operations, the PERQ simulates these lights by inverting small squares on the top of the PERQ screen. If the standard system window is displayed, the lights appear inside the title line.

Currently, the PERQ operating system uses three lights. The leftmost is used to show when the PERQ is doing a disk or floppy recalibration. This is done when the microcode is confused about where the heads are. Opening the floppy door during a disk operation usually causes this to happen. The light appears about 5/8 inches from the left margin.

The next light is used when the file system is attempting to fix up a file at run-time. This is needed occasionally when a file system operation has not completed normally. This light appears about 1-1/2 inches from the left margin.

The last light defined by the operating system is used for swapping. While a swapping operation (swap in or swap out) is in progress, this light is "lit." The swapping light appears about 2-1/4 inches from the left margin.

The Pascal compiler uses an additional light to indicate that the compiler is swapping its symbol table blocks between memory and the disk. (The compiler swaps symbol table blocks more frequently in systems with 256k bytes of main memory than in systems with 512k bytes or 1024k bytes of main memory. Therefore, the light is lit more often on minimum memory systems.) The compiler symbol table block swap light appears about 3 inches from the left margin.

10. Specifying a Filename.

The PERQ file system is described in detail in the PERQ File System Utilities Manual. This is a brief overview, a short introduction to path and filenames, a description of the various ways to specify a filename, and an explanation of the wildcard convention.

Overview of the PERQ File System

The PERQ file system has a hierarchical structure. This is reflected in the syntax of filenames. Files are stored on devices, which are divided into partitions. Each partition contains a number of files and directories; each directory may contain other directories and files.

Introduction to Path and File Names

The route that is traveled to reach a filename is called a PATH. A full path name specifies device, partition, directories, and filename, in that order. The syntax of a path name is:

device:partition>[directory>]filename

The brackets surrounding the "directory" part indicate that there may be zero to nine occurences of ">directory". Note that if you specify a directory, the brackets are NOT part of the syntax. To specify the current directory, or one of its subdirectories, you can omit the :partition> and the directory> syntax elements (see the description of the Path command in the PERQ Utilities Programs Manual).

1. Device names are usually assigned by the user. Examples of devices include floppy and hard disk. The syntax used to denote a device name is:

    device:

If you omit the device name, the system assumes the name of the device you booted from.

Each PERQ is given a name, which is usually the device name of the hard disk.

A device is accessable when it is mounted. See the description of "Mount" in the PERQ Utilities Programs Manual.

2. Partitions are set and named at device initialization time. You can modify partitions using the Partition program, described in the PERQ Utilities Programs Manual. Each device is divided into a fixed number of partitions;

the recommended size for a partition is 10,080 or fewer 256-word blocks. Files must fit entirely within a single partition as they cannot cross partition boundaries. Generally there are three partitions on a 12-megabyte disk and five on a 24-megabyte disk. Examples of partition names are BOOT and USER.

3. Directories are easy to create, destroy, rename, and move around. There can be multiple directories in a partition. These are denoted by the symbol > and can take names of up to 25 characters. There can be up to 9 directories listed in a full path name. The symbol:

..

is a convenient way of referring to a parent node in a tree. It goes up one node. An example of its use is:

SYS:BOOT>NEW>..>filename

This will look up filename in SYS:BOOT, rather than in SYS:BOOT>NEW>. The symbol

.

refers to the current directory.

4. Filename is the last thing specified in a file specification. A filename can have up to 25 characters.

The names of all of the directories and the filename cannot exceed 80 characters. You can include most special characters in a filename by surrounding the special characters with a single quote ('). For example, you could include the asterisk (*) in a filename by specifying '*'. Note that you cannot include the special and control characters described in section 4.

When you boot, the system takes as default device and partition the device and partition that you booted from. For example, the system might come up with a default of:

SYS:BOOT>

Ways to Specify a Filename

1. You can specify a full path name:

device:partition>[directory>]filename

the brackets, which are not part of the syntax, indicate that you can list up to nine directories. This is useful if you want to access a file on a different device from the one you're using.

2. You can use the default device that was determined at boot time:

:partition>[directory>]filename

Using this syntax enables you to look for a file in a different partition.

3. You can use the default device and partition that were set at boot time:

>[directory>]filename

The search then starts at the first directory specified.

4. You can just type:

filename

and the search begins at the current directory (which may be set by you, using Shell's PATH command). This can involve any device, any partition, and any directory in that partition.

Setting the default path doesn't affect the default device and partition used in forms 1, 2, and 3.

Along with the concept of a current directory, the file system provides a search list. This gives the user the ability to specify a set of directories to be searched, in a specific order, when a filename is specified. Refer to the SetSearch command description in the PERQ Utilities Programs Manual.

Wildcard Convention

A number of PERQ programs use a wildcard convention when looking up files. The wild cards are as follows:

* matches 0 or more characters.
& matches 1 or more characters.
# matches exactly 1 character.
'0 matches any digit.
'A or 'a matches any alphabetic.
'@ matches any non-alphanumeric.
'* matches *. Other wild cards can be quoted also.

There can be any number of wildcards in file specifications to programs that handle this convention. Examples of wildcard usage are:

Dir *.Pas

Dir &Boot*

The first command gives a directory of all files with a .pas extension. The second command gives a directory of

all files that have one or more characters followed by the characters "boot", followed by zero or more characters.

11. Default File Extensions.

An extension is a conventional sequence of characters that appears at the end of a filename. In the PERQ operating system there is nothing special about extensions. However, there are certain conventions that are used in the system. An extension is found by finding the last "." in the filename and taking the following symbols. Backup files conventionally have a "$" as the last character of their last extension; they take the form Name.Ext$. Following is a list of the standard extensions and how they are used.

.PAS - Pascal source files usually have this extension.

.SEG - The Pascal compiler produces a .SEG file when it compiles a .PAS file. .SEG files are used as input to the Linker and contain the code that will be executed when the program is run.

.RUN - Files produced by the Linker have this extension.

.MICRO - PERQ microcode files have the extension .MICRO. They can be used as input to the micro assembler, PRQMIC.

.BIN - The runnable version of PERQ microcode is contained in .BIN files. These files are produced by the micro placer PRQPlace.

.REL, .RSYM, .INT - These three file types are temporary files that are produced by the micro assembler and placer. They can be deleted after a .BIN file has been created.

.DFS - These files are used to communicate definitions between programs that may be in different languages, for example, between Pascal and microcode.

.CMD - A number of programs on the PERQ accept commands from a file as well as from the keyboard. Files that contain the commands usually have this extension.

.DR - In the PERQ file system directories are files. These files appear in a directory listing with the extension .DR.

.KST - Character set definitions are kept in files that have the extension .KST.

.MBOOT, .BOOT - When the PERQ is booted it reads the microcode interpreter and Pascal operating system from files that have the extension .MBOOT and .BOOT.

.ANIMATE, .CURSOR - Pictures used by utilities that are put into the cursor that follow the pen or puck.

.INDEX - An index to .HELP files.

.DOC - Formatted documentation has this extension.

.HELP - Files containing help about a subsystem use this extension.

12. Booting the Machine.

The PERQ can be booted in one of two ways. First, when the machine is powered up it will go through the boot sequence. Second, the machine can be booted by pressing the Boot button on the back of the keyboard. In either case the same sequence of events happens.

The boot sequence for PERQ has three steps. As each of these steps progresses, the Diagnostic Display, DDS, increments. The DDS is a three-digit number display. On earlier models of the PERQ, it is found on the inside of the machine behind the front cover; on later PERQs it's under the keyboard. If any step fails it is possible to look at the Diagnostic Display and see where in the boot sequence the failure occurred. The Fault Dictionary provides a list of the DDS values and explanations of their meanings.

In the first part of the boot sequence, microcode is executed out of a small ROM. This ROM covers the lower 2k of standard control store during this part of the boot sequence. This microcode runs a simple diagnostic on the processor and memory systems. If there are any errors, the microcode halts. The value in DDS gives the reason that the machine halted. Once these diagnostics have been passed, the microcode makes a decision about which device is to be used for booting. Currently, there are three possible boot devices.

1. The first alternative is booting from the hard disk. The microcode tries to boot from the hard disk.

2. The second choice of boot devices is a floppy disk. The microcode checks to see if there is a floppy in the floppy drive. If there is a floppy in the drive, PERQ will check to see if the floppy is a boot floppy. If so, the second part of the boot will be done from the floppy.

3. The third possible boot device is another PERQ. The microcode determines if there is a PERQ Link Board plugged into the I/O Option slot of the machine. If the board is plugged in, and there is another PERQ on the other end of the link, the booting PERQ will wait for commands from the link.

If all of these fail, then the DDS will contain an indication of what the error is. See the Fault Dictionary Manual for an explanation of the display number.

After the boot device has been chosen, the second part of the boot sequence can begin. In this part, the PERQ reads 3k words of microcode from the selected boot device. This microcode is in two sections, a more extensive diagnostic (VFY) and a system boot loader (SYSB). VFY attempts to verify that all of the CPU and Memory systems are working. Any failures that VFY discloses are displayed on the DDS.

If all went well, the microcode determines what set of interpreter microcode and system Pascal code is to be loaded. It does this by

checking to see if any key is being held down on the keyboard. If a key is being held down, that key specifies which boot is to be done. If no key is held down, the default boot will be done. The default is the same as holding down "a". Hold the key down until a pattern flashes on the screen. Any of the 26 alphabetic keys can be used to specify a boot. All lower-case characters cause a boot from the hard disk. Upper case characters cause a boot from floppy. If you type "Details/Boots" the Details program will provide you with a list of all of the valid boot characters. Once a boot has been chosen, the microcode interpreter and PERQ operating system are loaded into the PERQ. Control is then transferred to the third portion of the boot sequence.

In the third portion of the boot sequence, the interpreter microcode does any initialization that is needed and then starts to execute the PERQ operating system. The PERQ operating system also increments the DDS. If there were no errors during the boot sequence, the machine will be running and the DDS will read 999.

13. Run-time Errors.

When the operating system or a user program discovers an error condition, it raises an "exception" (see the "PERQ Pascal Extensions" manual for an explanation of exceptions). If an exception is not handled, it is given to the preliminary debugger, called Scrounge (see the "PERQ Utility Programs Manual"). First, a dump of the user state is produced and then the user is asked if he wants to debug. If not, the program is aborted and control returns to the Shell and any active command files are terminated. If the user decides to debug, the program can be continued after the point where the exception was raised. It is usually a bad idea to continue from uncaught exceptions. The user can also abort the program and return to the Shell.

14. Profiles.

Profiles can be used to tailor your PERQ. The profile is a text file which contains commands that define characteristics of certain utility programs. For example, a profile can direct the Login program to initialize the default path and searchlist. Each user of the system can have his own profile; the UserControl program can assign each user a profile file.

To create a profile file, copy DEFAULT.PROFILE to your own directory. For example:

```
COPY SYS:BOOT>DEFAULT.PROFILE SYS:USER>MYDIR>MYPROFILE
```

Now run UserControl and specify SYS:USER>MYDIR>MYPROFILE for the profile file. You can then edit the file and establish your specific conventions.

Each entry in the profile file begins with a number sign (#), followed by the name of the subsystem. For example, #LOGIN. The subsequent lines contain switches or data for the subsystem. The general format of a profile file is as follows:

```
#program1 <switches or input for program1>
          <more data for program1>
#program2 <switches for program2>
  ...
```

For example:

```
#Login /Path=Sys:User>Mydir
       /SetSearch=Sys:Boot>Library
       /CursorFunction=7
#RandomUtility /MaxSize=100
```

The format of each list of entries in the profile is defined by the utility program that uses the profile. You should read the documentation for a particular utility to determine whether it reads the profile, and if so, what entries can be included in the profile. You may use the profile in your own programs. See the Profile module in the Operating System Interface manual.

The Shell commands are specified in the #ShellCommands section of the profile. The format for each line in this section is:

<implementation><useDefault><setDefault><screensize><cmdname>

Where <implementation> is the command line Shell issues to execute the command. <useDefault> tells whether to use the default file name if no file name argument is specified for the command. <setDefault> specifies whether to set the default file name if an argument is provided. <screensize> can limit the screen to less than full size. <cmdname> is the name that will be used to invoke the command. This name and the rest of the line is printed when a

"?" is typed so the additional comments are provided to explain the command.

PERQ Utility Programs Manual

Diana Connan Forgy

Donald A. Scelza

Brad A. Myers

Bob Amber

This manual provides an introduction to the use of the PERQ Utility Programs.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

1. Preface: Notation Conventions.

This manual describes the PERQ Utility Programs and commands to the Shell. The command descriptions are in alphabetical order regardless of whether the command is implemented as a Pascal file or through the Shell directly (The Introductory User Manual describes the Shell.)

The descriptions of the commands given in this manual observe the following notational conventions:

- Lowercase text indicates a variable whose actual value is determined when the command is entered
- Square brackets ([]) indicate optional entries in a command line. Note that when an option is used, the brackets are not part of the syntax.
- The circumflex (^) indicates the control key
- CR indicates carriage return
- SHIFT indicates the shift key

2. Introduction

This manual describes the PERQ Utility Programs and explains their use. As required, the manual references online or other Three Rivers Computer Corporation publications.

Most of the commands described in this manual are implemented as Pascal files, but some are implemented directly by the Shell.

To issue a command, type a command line in response to the default PERQ prompt. You can also use a pop-up menu, as described in the PERQ Introductory User Manual, to issue Shell commands.

If you elect to use a pop-up menu, the menu displays all of the valid commands. Use the pointing tablet to specify the selection. The FLOPPY and FTP utilities also permit you to enter their respective commands through pop-up menus; a press of the pointing tablet in response to either of the prompts for these utilities displays a menu for the utility commands.

If you type a command line, the line consists of a command name, input and output arguments as needed, optional switches, and a line terminator.

The command name describes the action the system performs or the name of the utility that performs the action. When submitting a command that is implemented in the Shell, you need not type the entire command name; you can abbreviate the command name to the number of letters unique to other command names. For example, the Mount command can be abbreviated M because it is the only command within the Shell beginning with that character. You can abbreviate the Path command to Pat, but not Pa because the Shell includes the Pause command. The question mark command (see section 3) lists all commands. You can modify the list by changing your profile file.

Input and output arguments further define the command action. The input and output arguments are usually file specifiers. Some commands require arguments as part of the command line. Other commands use the default file as the argument if you do not supply one.

If you neglect to supply a required argument, the utility prompts with a few words indicating the general nature of the missing argument. For example, the Rename command performs as follows:

```
>RENAME
File to rename:  OLDFILE
Rename OLDFILE to:    NEWFILE
```

The single line format for the above command is:

```
RENAME OLDFILE ~ NEWFILE
```

or

RENAME OLDFILE NEWFILE

You can mix formats; the utility prompts for whatever you omit. For example:

```
RENAME OLDFILE
Rename OLDFILE to:  NEWFILE
```

There are no defaults for prompts. You must supply a response. However, your response to the prompt can request help (type /HELP or press the HELP key). If you request HELP, the utility displays specific information and then exits to the Shell.

If you neglect to supply an argument to a utility that uses the default file (for example, the Editor or Linker), the utility appends the default file name to your command. For example, if the default file is sys:user>myfile, typing the command:

EDIT

is the same as typing:

Edit sys:user>myfile

If a command requires an input and an output argument, you can specify the arguments as either

INPUT~OUTPUT

or

INPUT OUTPUT

You can separate the input and output arguments with multiple spaces; only the initial space delimiter is relevant.

However, if a command accepts multiple input or output arguments, you must separate like arguments with commas (,) and distinguish input from output with the tilda character (~). For example:

input1,input2,...inputn ~ output1.output2,...outputn

If a command accepts multiple input arguments and no output arguments, you must separate the arguments with a comma (,).

Switches modify the action of the command and therefore must follow the command specification. An exception is the Help switch (either type /HELP or press the HELP key); when specified before the command, the Help switch supplies general information and when specified after a command, the Help switch provides specific information. Switches always start with a slash (/) and are generally optional. If a switch accepts a parameter, specify the parameter after the switch, but preceded by an equal sign (=). For

example:

```
/switch=parameter
```

The effect of a switch is global; regardless of where the switch appears on the command line, it has the same effect. A switch applies to every argument. If a command accepts multiple input or output arguments, no switch applies to one and not another argument. Some switches are mutually exclusive (for example, /ASK and /NOASK). If you specify a switch that conflicts with a previously specified switch, the last occurence has precedence. Likewise, if you change parameters by specifying a switch multiple times, only the last occurence has an effect.

All of the commands and utilities described in this manual accept the /HELP switch (either type /HELP or press the HELP key) to provide general information. Note that /HELP overrides all other switches; the utility displays specific help and then exits.

Carriage return is the line terminator for all commands.

Rather than typing a command line to initiate and direct a utility, you can use a user command file. A user command file is a text file containing a series of commands interpretable by the various utilities.

A user command file is a sequential file containing a list of utility specific commands. Rather than typing commonly used command sequences, you can type the sequence once and store it in a file. The user command file is specified in place of the command line(s) normally submitted to the utility.

To inititate user command files, replace the command line for a utility with a file specifier, preceded by an at sign (@). The utility requesting input then accesses the specified file and starts to read and respond to the commands contained within it. For example, to initiate a file of FLOPPY commands, type the following in response to the FLOPPY prompt:

```
FLOPPY>@FLOPPY.CMD
```

The FLOPPY utility accesses the file and then executes the commands contained within the file FLOPPY.CMD. The default file type for user command files is .CMD. Thus the above command line could also be typed as follows:

```
FLOPPY>@FLOPPY
```

You can nest user command files by simply specifying @file within the user command file. Also, the last line of a user command file can invoke another command file and recursive use is permitted.

To comment a user command file, start the comment line with an exclamation mark (!). The exclamation mark can appear anywhere on a line in a user command file, but remember that the system ignores all characters after the exclamation mark.

3. Question Mark: ?

The question mark command lists all the commands implemented in the Shell. Additionally, the command provides some information on what each command does and how it is called.

4. Append

Append copies one or more files to the end of an existing file. The command accepts a list of input files, separated by commas, and puts each file on the end of the first. For example, the command line:

```
APPEND file1,file2
```

puts file2 on to the end of file1. The append operation is successive. For example, the command line:

```
APPEND file1,file2,file3
```

first puts file2 on to the end of file1 and then puts file3 on to the end of file1.

5. Bye

Bye logs you off the PERQ. Type:

```
BYE
```

and it types your name, the date, and time of logoff. Login then loads and you or some other authorized user can log back on.

The command

```
BYE OFF
```

logs you off and turns off your machine. OFF may be disabled on your PERQ; if so, call Three Rivers Computer Corporation for details on reenabling it. OFF requires some special microcode which should be available on your machine. If not, Bye will log off and request that you power down the machine by hand. In this case, ^C will cause Login to run.

The command

```
BYE WAIT
```

logs you off the machine and sends the hard disk heads to the center of the disk (the highest disk address). Bye then prints a message and waits for power down or ^C. When shipping a PERQ, always type BYE WAIT to position the disk heads at the center of the disk.

The command

```
BYE HELP
```

describes the options available for Bye. You can specify all of the options (OFF, WAIT, and HELP) as switches.

It is very important to log off using BYE before powering down the machine. If you do not, certain temporary disk files will not be deleted and they will stay around using disk space until you run the Scavenger (see below).

6. Chatter

Chatter allows a PERQ to act as a terminal, using an RS232 line for communication. It is invoked by typing:

CHATTER

While in Chatter, some special functions can be invoked by typing ^R, the function's code letter, and CR. These include

S to save everything that comes from the remote computer in a file.

T to transmit a file across RS232, as if it were being typed at the keyboard.

C to close a file after doing a Save.

B to change the RS232 baud rate. The default when Chatter comes up is 4800 baud.

Q to quit Chatter and return to the Shell. This doesn't log off or disconnect the remote host.

This menu will appear at the very top of your screen to prompt you after you've typed ^R.

While in Chatter, if you find that the characters you type are not being echoed properly, one of two things may be wrong: the baud rate may be inappropriate or your RS232 cable might not be connected properly.

Chatter cannot be used as a half duplex terminal since it does not echo characters locally.

7. PERQ Pascal Compiler

The PERQ Pascal compiler translates your Pascal source program into a .Seg file that can be linked and run. There are several ways to invoke the compiler and several options that you can use with it. The command line takes the form:

```
COMPILE [inputfile] [~] [outputfile] [/switch(es)]
```

Examples of legitimate compiler calls include:

```
COMPILE Program.pas

COM ProgramX
    (Note that the .pas extension is implicit;
     if ProgramX does not exist, the compiler
     looks for ProgramX.PAS.)

COMP Program2~Program1
    (creates the output file Program1.SEG)

COM
    (compiles the default file)

COM /symbol=32
    (compiles the default file with 32 symbol
     table blocks)
```

If you want to compile a program immediately after editing it, you need not specify its name since the Shell remembers the last file edited, compiled, linked, or run.

Certain switches may be included in the source program text. See the PERQ Pascal Extensions Manual for more detailed information on switches and other compiler features.

8. Copy

Copy creates a new file identical to the specified source. Its command syntax is:

COPY SourceFile[~]DestinationFile

Copying works across devices and partitions (see the PERQ Introductory User Manual for details on devices and partitions). You can also specify the non-file-structured devices CONSOLE: and RS:. If you copy a file to the RS232 interface, by default the interface is driven at 9600 baud. To change the baud rate, run the SetBaud program (see below).

The source file for Copy may contain wild cards (for a description of the wild cards, see the PERQ Introductory User Manual). If the source contains wild cards, the destination must contain the same wild cards in the same order. In this case, Copy matches all files in the directory with the source pattern. For each match, the part of the source file name that matches each wild card replaces the corresponding wild card in the destination. As an example, for the command:

COPY foo*.abc# anotherdir>*baz.rmn#z

the input file "FOOZAP.ABCD" would be copied into a new file named "anotherdir>ZAPbaz.rmnDz".

If wild cards are used, Copy asks for verification of each file copied. This can be disabled with the switch "/NOASK" or enabled with "/ASK". The latter is the default. Copy also requests confirmation before overwriting an existing file. You can override this action with the /NOCONFIRM switch. (Note that the /NOCONFIRM switch implies the /NOASK switch.)

Wild cards are not allowed in the directory part of the source file. However, Copy uses the search list to try to find the source. Note that this is different from Rename and Delete which always look in only one directory.

When the source file name contains no wild cards, the destination file name may contain, at most, one occurance of the wild card "*". In this case, the non-directory part of the source replaces the "*" in the destination. For example,

COPY sys:Boot>newOS>myprog.Pas dir3>new.*

would copy the file "sys:Boot>newOS>myprog.Pas" into a new file named "dir3>new.myProg.Pas". This is most useful when you want to copy a file from one directory to another with the same name. For example,

COPY dir1>prog.Pas *

copies prog.pas from the directory "dir1" into the current directory.

If there are no wild cards in the source, an attempt to include in the destination name other wild card characters, besides the single "*" discussed above, may lead to problems later. These extra wild cards will be treated as simple literal characters. Because a file name with wild cards in it is hard to specify, the Copy program requires confirmation before creating a file with wild cards in the name.

Copy is a useful command if you'd like to edit your own copy of a file without changing the original one. (The Editor provides you with the ability to do this too, but copying can give you an additional safeguard against unintentional changes to the source.)

The names of SourceFile and DestinationFile may be identical; however if they are identical a new file will not be created. If you want two files you should use nonidentical names.

If the file you are copying to already exists, Copy requests confirmation before overwriting it. This can be disabled by using the switch "/NOCONFIRM" or enabled using the switch "/CONFIRM". NOCONFIRM also sets NOASK. If an error is discovered and wild cards were used, Copy asks the user whether to continue processing the rest of the files that match the input. This confirmation is required no matter what switches were specified.

A final switch is "/HELP" which describes the function of Copy and the various switches.

9. The Preliminary Debugger

The current operating system includes a simple debugger. When the user types ^SHIFT-D or an uncaught exception is discovered, a dump of the user stack is shown. This has the form:

```
Control-shift-D dump
Debug at      108 in routine 7 in IO_PRIVA.
Called from   214 in routine RANDOMWI (8) in WIPEWIN.
Called from   395 in routine WIPEWIN (0) in WIPEWIN.
Called from   149 in routine 0 in LOADER.
Called from   222 in routine 1 in SYSTEM.
Called from   520 in routine 0 in SYSTEM.
```

First, the reason for taking the dump is shown. Next is a trace of all the procedures on the stack. Each line shows the location in the code, the routine that location is in, and the module which contains that routine. The location is the QCode offset from the beginning of the procedure. You can use QDis to try to associate this with the corresponding place in the source. When the debugger can find the procedure name, it is printed followed by the procedure number. At other times, only the routine number will be printed. You can count the procedure (and exception and function) headers in the module source file to determine which procedure it is.

When counting procedures, start with one for programs and zero for modules. The main body of a program is its procedure zero. Exported and forward procedures are counted only once, where the name first appears. Internal (nested) procedures are counted exactly like other procedures.

A note about procedure names: These names are always truncated to eight characters and converted to all uppercase. To get the names, the debugger examines the Seg file for the module. If the Seg file found is not the one that was loaded, the procedure names will be wrong. The procedure numbers will always be correct, however. For the procedure names of system modules, the system run file is checked to find the Seg file name. This Seg file is used to get the procedure names. If the system run file or the system Seg files are not accessible, the debugger will not be able to print the procedure names for system routines.

After the dump is printed, you will be asked if you want to debug. (If a dump is printed due to a ^SHIFT-C, no debugging will be allowed). If you answer no, the program will be continued if it was called from ^SHIFT-D, otherwise it will be aborted and control will go back to the Shell. If you decide to debug, the debugger will print something like:

```
Scrounge, V0.10

  Now at routine KEYINTR (7) in IO_PRIVA
  There are 5 local words, 0 argument words, and 0 result words
```

```
Debug>
```

Now you can use the debugger's commands to investigate your program. Notice that the debugger goes to more effort to find the procedure names than the original dump, so that if a procedure name was not printed at first, going into the debugger may get it displayed.

The debugger does not know the types or sizes of variables, but it does know the number of words allocated for locals, arguments and results. Note that the compiler may generate temporary variables which are included in the local count.

When a debugger command is invoked which takes an offset, you can type a number which is the offset of the word to print. Zero is the first word. If you type -1, all the words in the current context will be printed. For example, to an "a" command, all the arguments will be printed for -1. If you type -2, the debugger will request the first and the last offsets to print. In this manner, a range of values can be printed. No checking is done to make sure that the offsets typed are in range; if a number is out of range, some random data will be printed. Data is printed in the form:

```
[ 7] ( 5053^) = 6
```

where 7 is the offset in the current procedure, 5053 is the offset from the bottom of the entire stack and 6 is the value in that location.

A note on counting variables: Imagine that your procedure was defined with the following variables:

```
    var a,b,c: Integer;
        d: Char;
```

When a list of variables are defined in the same statement, they are allocated in reverse order so the first word is the variable "c". The second word is "b", the third "a" and the fourth is "d". This is true for local and global variables and for records. Note that if you had declared the variables as:

```
    var a: Integer;
        b: Integer;
        c: Integer;
        d: Char;
```

then "a" would be the first word, "b" is the second one, etc. This does not hold for procedure parameters, however, where the variables are always stored in exactly the order declared.

The debugger's commands are:

? Print a list of all the commands.

x Set the radix. All integers are normally printed in signed decimal. With this command, you can specify any radix from 2 to 36. If the radix is negative then all output will be unsigned. If it is positive then output will be signed. Note that this radix is only for output; all input is still in signed decimal.

> Up level. Move up one level in the stack towards the top of the stack. To investigate the variables of a procedure, move up or down the stack until that procedure is reached and then it can be investigated.

< Down level. Move down one level towards the bottom of the stack. When entering the debugger, the current procedure is set at the top so this command has to be used first.

^ Dereference. Dereference any pointer in memory. This takes a segment number and an offset. For a variable parameter or pointer variable, the offset is first and then the segment number.

t Top of Stack. Move to the top of the stack.

b Bottom of Stack. Move to the bottom of the stack.

c Current. Shows number of words for arguments, returns and locals for the current procedure.

d Display Stack. This command reprints the original dump. Some additional procedure names may be printed. In addition, the procedure where you are debugging is marked with "<**>".

l Local. Examine the local data. The debugger reprints the number of local words. You can type the offset of the word you want to see.

a Argument. Examine the arguments to a procedure or function.

e Exception. Examine the arguments (parameters) to an exception.

r Returns. Examine the return values from a procedure.

g Globals. Examine the globals for the module or program that the procedure is in. When the g command is given, the module or program name is printed. If it is a program, the debugger asks if you want to skip input and output. These are the two file variables that are defined for every program and take up space at the top. If you answer yes to this question, you do not have to worry about the space for them when counting variables. Unfortunately, there is no way to examine data in modules that do not have procedures on the stack.

m Mode. The debugger cannot know the type of data, but if you know, you can tell the debugger. The mode command lets you specify the output mode for data. When you type the Mode command, it prints the current mode and asks for a new one. If you type "?" at this point, a list of the options is printed. They are: i=integer, s=string, c=char, B=Boolean, b=byte. Notice the case sensitivity of the arguments. When the mode is string, the debugger cannot print a range or all data since it cannot know how much memory was allocated to hold the first string. In this case, if the -1 argument is

given, the offset is assumed to be zero. When printing strings, the length is printed first. When printing bytes, the radix specified still holds (although they will always be unsigned). For bytes and characters, offsets are still in terms of words; the debugger prints both bytes in the word specified.

s Stack. This command permits display of words anywhere on the program stack. Detailed knowledge of the compiler's memory allocation is necessary to utilize this command so it is generally not useful.

q Quit. Exits the debugger and aborts the program that was running. This returns control back to the Shell. It requires confirmation.

p Proceed. Exits the debugger and resumes the program executing. Note that this command allows you to resume from uncaught exceptions but this is not recommended. In this case, confirmation is required. If the debugger was entered through ^SHIFT-D, then no confirmation is required.

If an exception is raised inside the debugger, the debugger aborts immediately and exits to the Shell. In the debugger, ^C and ^SHIFT-C both cause immediate exit to the Shell also. ^SHIFT-D is disabled while inside the debugger. Also, the HELP key does not work while in the debugger; use the debugger question mark (?) command.

10. Delete

The command:

DELETE FileSpecification[,file2,...filen][/switch]

irrevocably destroys the specified file(s). It deletes the file's name from the directory and places the blocks it occupied on the free list, thus making those blocks available for use by other files.

Wildcards may be used in the file specification, but not in a directory part. See the section entitled "Specifying a file name" in the PERQ Introductory User Manual for details on PERQ wildcard conventions.

The valid switches for Delete are:

/CONFIRM
asks for verification before deleting a file. It's the default when you use a wildcard unless you use a PopUp menu (see below).

/NOCONFIRM
is the default when you specify only a filename without a wildcard.

/HELP
provides some online documentation.

Delete also allows you to select the files you want to delete by using a PopUp menu (see the section on "PopUp Menus" in the PERQ Introductory Users Manual). To get a PopUp menu, type Delete followed by a carriage return (no arguments). Delete will prompt with

File to delete or press for Menu:

at this point you can simply press the pen or puck for a menu. You can also type a file name followed by switches. For example, you might type

:boot>myDir>*.TMP/confirm

and then press down on the pen or puck. You should not type a carriage return before pressing.

The menu displayed when you press contains the files that match the file pattern. If no pattern is typed, all the files in the current directory are listed. Simply select in the menu the files that should be deleted. Unlike the menu for the Shell, the Delete menu allows you to select multiple files. All selected files are marked by reverse-video. These are the files that will be deleted. You can de-select a selected file by simply pressing on it again. Since the number of files that can be displayed is limited, scrolling is

provided when the the number of files matching the file specification is large. Use of the scrolling feature is described in the PERQ Introductory Users Manual.

Once you have selected all the files you wish to delete, move the cursor to the lower right corner of the menu to the spot with the "x" in it. The cursor should change to a large exclamation point. When you press here, all the deletes will take place. If, however, you press outside of the menu before pressing here, no deletes will take place and the program aborts.

When using a PopUp menu for deletion, the default is NOVERIFY. Thus all selected files are deleted when you press the exclamation point. For added safety, you can still specify the /VERIFY switch (as shown above) to cause the system to ask for confirmation on each of the selected files before deleting it.

11. Details

The Details command provides some information about the current state of your PERQ. Typing DETAILS /HELP will get you online documentation. The Details command line is of the form:

DETAILS [/switch]

Valid switches are:

/USERNAME	Name of current user
/USERID	ID of current user
/MEMORYSIZE	The size of memory
/PARITYERRORS	Parity error information
/PROFILENAME	Name of profile file for the current user
/PARTITION	Names of all devices and partitions known (includes the number of free blocks in each partition)
/LOADEDPROFILE	Profile information that is cached in memory
/SEARCH	Prints the current search list.
/SHELLNAME	Name of current Shell runfile
/SHELLINFO	Shell specific information
/DISKSIZE	Size of hard disk
/TIME	Gives current date and time
/PATH	Gives current path, default partition name and default device name
/LASTFILE	Default file for Edit and Compile
/BOOTCHAR	Character used for booting
/BOOTS	Valid boot characters
/SWAP	Whether swapping is enabled or not and to where
/IOERRORS	A count of how many times each of the IO errors occurred since the last boot
/ALL	Displays all of the above information
/HELP	For online documentation

These may be abbreviated to as many characters as are unique. If you don't specify a switch, a selection of the available information is typed. If you specify *, all information is typed.

"Details /Partition" is very useful since it tells how much free space there is in all the partitions.

See the PERQ Introductory User Manual and the PERQ File System Utilities Manual for details on Partitions, Search lists, Shell, Path, and Boots.

12. Directory

The Directory command provides an alphabetical list of files in a directory. You can display the directory listing at your terminal or you can specify that the Directory command write the listing to a file. The form for the command line is:

DIRECTORY [dirSpec>][fileSpec]{/switch}[~][outputfile]

If it is invoked without a switch, all files in the current directory will be listed. To write the output of a Directory listing to a file, specify an output filename.

If there are wild cards, the dirSpec part is matched against all directories and the fileSpec part is matched against all files in the directories that matched dirSpec. Wild cards are described in the section "Specifying a file name" in the PERQ Introductory User Manual. Wild cards are not allowed in partition or device names.

Examples of usage include:

DIR
lists every file in the current directory

DIR *>*
shows all files in all directories starting with the current directory and including all subdirectories.

DIR/HELP
gives online documentation

DIRECTORY :BOOT>x*>*.run~run.list
looks in the Boot partition for all the run files in directories whose names start with "x" and writes all of these names into the file "run.list".

DIRECTORY Program*
tells you what files beginning with "Program" are in the current directory, e.g., Program.pas, Program.seg, Program.run.

DIRECT *zing*
lists all files with "zing" in their names.

DIR Program*/SIZE
lists files beginning with "Program" and tells how much disk space each occupies.

The following are the switches available for use with this program:

/HELP
types online documentation.

/FAST

prints a short directory. This is the default.

/SIZE

tells you how many blocks and bits are in each file.

/ALL

provides the following information about each file:

Number of Blocks
Number of Bits
Kind of file
Creation date
Last Update date
Last Access date

/LISTDIRECTORIES

When doing a multi-directory listing, only the directories that have valid matches for the fileSpec are printed. This switch tells Direct to print all directories that match the dirSpec even if they do not contain any matches for fileSpec.

/ONECOLUMN

tells Direct to print all files in one column. This is the default when the output goes to a file.

/MULTICOLUMN

tells Direct to print files in four columns. This is the default when doing a FAST directory to the screen. This switch does nothing if SIZE or ALL is specified.

/DELIMITER

when used in conjunction with an output file specification, writes filenames as

name | name

into a file. This is useful for creating command files.

/PARTITIONS

gives information about all partitions after the files are listed.

/SORT=option

specifies the method in which the directory

is sorted and displayed. This switch accepts the following options:

NOSORT - do not sort the directory. In this case, all the files are listed in essentially random order. This switch is useful when swapping is turned off and Direct does not have enough memory to sort the file (for example, when running from the floppy).

NAME - sort by the name of the file. This is the default.

SIZE - sort by file size. This operation lists files in decreasing order, with the largest file first.

CREATEDATE - sort by creation date. The most recent file is listed first.

ACCESSDATE - sort by last access. For this function, access is defined as the last read operation performed on the file.

UPDATEDATE - sort by last update. For this switch, update is defined as the last write operation performed on the file.

13. DirTree

DirTree gives a graphic representation of the filesystem's tree structure. It erases the screen and then displays a tree of all the directories starting from the root directory on the left. Any directory can be specified as the root of the tree. The default starting place is the default device. In this case, DirTree displays all the partitions, and then all the directories in each partition, and then all the subdirectories, etc. Lines are drawn from each directory to its parent. If a directory is supplied, DirTree simply starts the search from that directory.

If the specified, or default, root directory contains the current directory, DirTree highlights the current directory with reverse video. You can then change the path by moving the cursor to the desired directory and pressing the pen or puck. This is equivalent to issuing a Path command; DirTree permits you to change your current directory.

Typing any character or pressing in an area that does not contain a directory exits DirTree.

If the directory is too deep to fit on the screen, DirTree puts an asterisk (*) on the right of the parent. You can reinvoke DirTree with this directory as the root to see more of the tree structure.

DirTree accepts three switches: /WAIT, the default, enables pressing to select a new path; /NOWAIT disables pressing to select a new path (DirTree simply displays the tree structure); and /HELP for online documentation.

14. Dismount

The Dismount command detaches devices from the filesystem. The argument to Dismount is HARDDISK (H) or FLOPPY (F). Note that these names are used no matter what name the device was given when it was partitioned.

Once a device has been dismounted, it can no longer be accessed until it has been mounted again. It is very important to Dismount filesystem floppies before removing them from the drive.

See the section on "Mount" for more information.

15. Editor

The Editor is used to create or alter any text file on the PERQ. You will probably use it very frequently, so it's a good idea to become familiar with it as soon as you can. It has its own online documentation (run Edit and press the HELP key) and manual (The Editor User's Guide), so this description is brief. Three common uses of the editor are discussed here. The command line for Edit is:

EDITOR FileSpecification

or

EDITOR/replay

If the switch is left out Edit will assume that you want to edit the default file name remembered by the Shell. The /REPLAY switch is useful when disaster occurs during an edit session; you specify the switch, redo the edit session, and stop just before the disaster. Refer to the Editor User's Guide for details.

The Editor does extension completion on the file name specified. If the file to edit is FOO.PAS, it is only necessary to type FOO. The extensions that the Editor knows about, in order tried, are: Pas, Micro, Cmd, and Dfs.

The Editor signals the end of a file with a solid, left pointing triangle.

Three common uses of the Editor are:

1. To create a new file: Invoke the editor with the name of your new file. The screen is cleared to give you a blank page to write on. Type I to insert the text that you want to type in. When you're finished,

   i. Press the INS key (upper lefthand corner of keyboard). This is important; it's the only way that what you typed in will be saved.

   ii. Type Q. The screen is cleared again and you are prompted with a list of alternatives. See the Editor documentation for details on these.

2. To make changes to an existing file: Invoke the editor with the name of an existing file. Work with the Editor User's Guide at your side until you're comfortable with the functions available.

   If you find that you've made changes to a file that isn't yours or that you've done irreparable damage to one that is, don't panic - if, after typing Q you type E, all of the changes you've made will be ignored.

If you'd like to save your changes but don't want to alter the source file, you can type W after Q to make a new file.

3. To read a file at your leisure: you can EDIT it, reading and scrolling at your own pace. To safeguard against your having made any accidental changes to the file, type E after Q.

16. ExpandTabs

ExpandTabs simulates tabs in every 8th column by replacing tabs in the input file with the correct number of spaces. ExpandTabs is used when the input file was written for another system and put onto a PERQ, which does not support tabs. Its command line takes the form:

EXPANDTABS SourceFile DestinationFile

Note that the ExpandTabs command does not accept the Help switch.

17. FindString

The FindString command searches through a number of files for a particular string. There are two modes: /CONTEXT; and /NOCONTEXT. In /CONTEXT mode, FindString prints leading and trailing characters for each occurrence of the specified string. In /NOCONTEXT mode, FindString prints only the first occurrence of the specified string and does not print leading or trailing characters. The default mode is to print leading and trailing characters for each occurrence (/CONTEXT).

The first argument to FindString is the string to search for. If you want to include a space, comma (,), or slash (/) in the search string, you must precede it with a single quote ('). The next argument is the file pattern to match files against. The remaining arguments are optional. You can direct FindString to write the occurrence(s) to a file by specifying an output file. You can also specify a switch to show context, ignore upper and lower case, or request help. Thus, an example command line is:

```
FindString screen, :boot>os>*.pas~screen.users/nocontext
```

The above command directs FindString to search all files with a .PAS extension in the OS directory of the BOOT partition for an occurrence of the string screen. FindString writes the output to the file "screen.users". By default, case is not significant (in the example above, Screen matches screen). You can force FindString to match upper case characters by specifying the /CASESENSITIVE switch. FindString also accepts the /HELP switch.

18. Floppy

The FLOPPY utility formats, tests, reads, and writes RT-11 format and filesystem floppy disks. FLOPPY can also format filesystem floppies. You can use FLOPPY to transfer files between the hard disk and the floppy disk. The command line is:

FLOPPY [command][/switch(es)]

To execute a single FLOPPY command and return control to the Shell, enter a command on the command line.

To execute multiple FLOPPY functions, type FLOPPY and press return. In this case, the utility prompts with FLOPPY>. You can then enter commands or use the pop-up menu.

Some FLOPPY commands require confirmation. This confirmation comes from the keyboard even if a user command file is in use. If you use the FAST command (see below), no confirmation is required. The Zero and Format commands, however, require an explicit /NOCONFIRM switch to override the confirmation request.

While FLOPPY is processing a command, a "hand" cursor moves down the right margin of the screen. When it has reached the bottom, your operation is complete. (With small files, the cursor does not reach the bottom of the screen.)

The wild card handling in FLOPPY is more restrictive than the operating system's. FLOPPY only accepts wildcards for the DELETE and DIRECTORY commands (see the respective command descriptions). In addition, the only wild card available is the asterisk (*) and it can only be used by itself in either the filename or extension part of a file specification (or both).

Current FLOPPY commands and their switches are:

COMPRESS: Coalesce free space on the floppy. This moves files so that all the unused blocks are at the end of the floppy. COMPRESS does not accept input or output arguments, but has a switch which turns verification on or off. The default is verify; if this is on, COMPRESS checks every transfer to assure there are no errors. To COMPRESS in verify mode, you need not specify the switch, since /VERIFY is the default. The /NOVERIFY switch overrides the default.
NOTE: This command takes a long time and cannot be interrupted once it starts.

DELETE: This command deletes a file or multiple files on the floppy. To delete multiple

files, you must separate the filenames with a comma (,). By default, the DELETE command requests confirmation before deleting a file. You can override this by specifying the /NOCONFIRM switch. Wildcards are allowed.

DIRECTORY: This command lists the files contained on a floppy and optionally writes the directory listing to a file. If you specify the DIRECTORY command with no arguments, it lists all the files contained on the floppy. If you specify an input argument, DIRECTORY lists those files that match the specified filename. If you specify an output argument, DIRECTORY writes the listing to a disk file with that name, but does not display the filenames on the screen. By default, the DIRECTORY command prints (or writes) a full listing. To override this and print only the file names, specify the /SHORT switch.

DUPLICATE: This command copies the contents of one floppy onto another. The command creates a set of scratch files on the hard disk, copies the scratch files back to the new floppy, and then deletes the scratch files from the hard disk. To retain the scratch files on the hard disk, specify the /NODELETE switch. By default, the Duplicate command creates a double-sided floppy. To override this and create a single-sided floppy, you must specify the /SINGLESIDED switch. Remember that the blank floppy to be duplicated must have been formatted prior to invoking Duplicate.

FLOPPYGET: This command copies the contents of a floppy disk to the hard disk. You can optionally specify a hard disk file name. The Floppyget command accepts the /SINGLESIDED switch to permit you to specify a single-sided floppy.

FLOPPYPUT: This command copies the disk files created by Floppyget onto a floppy disk. By default, Floppyput deletes the disk files after copying them to the floppy. You can specify the /NODELETE switch to override this action. For single-sided floppies, you must use the /SINGLESIDED switch.

FAST: When issued as a command, FAST turns off requests for confirmation for all subsequent commands except Format and Zero. Use

the FAST command only in conjunction with command files.

FORMAT: This command formats a floppy disk and destroys its current contents. The FORMAT command accepts the following switches:

/DblDensity=Yes or No (default is no)
/Noconfirm (override request)
/Singlesided=Yes or No (default is no)
/Test=Yes or No (default is no)

GET: This command copies one or more floppy files to the hard disk. In its simplest form, you specify only an input filename to copy from the floppy to the hard disk. GET then copies the floppy file to the hard disk using the same filename. If the filename already exists on the hard disk, GET requests confirmation before overwriting it. Other forms of the command permit you to name the hard disk file. For example:

```
GET floppyfile~harddiskfile
```

To specify multiple files, use the following construct:

```
GET f1,f2,...fn~h1,h2,...hn
```

Like the COMPRESS command, GET takes the /VERIFY and /NOVERIFY switches. The default is /VERIFY.

The GET command requires confirmation before overwriting a file on the hard disk. You can specify the /NOCONFIRM switch to override this action.

HELP: When issued as a command, HELP provides general information for the FLOPPY utility. You can get specific help by using the /HELP switch. Note that HELP and /HELP override any other commands or switches; FLOPPY displays the requested help and then reprompts.

PUT: This command parallels the GET command, but transfers a file or files from the hard disk to the floppy disk. PUT also requires

confirmation before overwriting an existing file on the floppy (override this with /NOCONFIRM) and, like GET, accepts the /VERIFY and /NOVERIFY switches.

QUIT: Exits the FLOPPY utility.

RENAME: Changes the name of a floppy file. If the new filename already exists, RENAME asks for confirmation before overwriting it. You can override this with the /NOCONFIRM switch.

DENSITY: This command tells the user whether the floppy is single or double density.

TYPE: This command types a floppy file on the screen. The "hand" cursor tells how much of file has been typed. When the Type command finds a formfeed character (^L) in the file, it waits after displaying a screenful of text. This enables the user to read the page before the screen is erased and the next page displayed. To continue reading, type ^Q. You can disable this wait feature by specifying the /NOWAIT switch. The Type command displays a solid, left pointing triangle when it reaches the end of the file.

COMPARE: This command takes a disk file and a floppy file (in that order) and ensures that all bytes are identical. You can specify multiple disk files (input arguments) and multiple floppy files (output arguments) for the comparison. If you specify multiple arguments, you must separate like arguments by a comma (,) and delimit input form output arguments with an equal sign (=). COMPARE prints a message for every block that contains a difference.

ZERO: This command creates a new directory on the floppy. By default, the directory matches the number of sides on the floppy. You can override this by specifying the /SINGLE-SIDED switch. The Zero command always requests confirmation before creating the directory. You can override this only by specifying the /NOCONFIRM switch. In the process of creating the new directory, the ZERO command destroys the contents of the current floppy. Use the ZERO command after formatting a floppy (see the FLOPPY FORMAT command description in this section) or whenever you wish to destroy the current

content of a floppy.

Note that when you issue the ZERO command, you irrevocably destroy the current contents.

19. FTP

The FTP utility copies files across the RS232 link to another PERQ or any other computer that supports the FTP protocol. You can also use FTP to transfer files across an ETHERNET connection. The command line is:

FTP [command][/switch(es)]

To use the RS232 line, the two machines must be connected by an RS232 cable, which goes into their RS232 ports (located between the cables for the display and keyboard in the lower lefthand corner of the back of the PERQ).

To use an ETHERNET connection, you must first assign the Ethernet addresses in the file SYS:BOOT>ETHERNET.NAMES. The format of this file is the node name followed by a unique six byte address. The first three bytes are the Ethernet address blocks for Three Rivers Computer Corporation (02 1C 7C in hexidecimal) and the second three bytes identify the individual nodes. You can include up to ten nodes in the ETHERNET.NAMES file. The basic, and recommended format follows:

```
PERQ1     540  31744  1
PERQ2     540  31744  2
      .
      .
      .
PERQ10    540  31744  10
```

You can then issue the FTP Address command and specify the local and remote names. If the specified names exist in ETHERNET.NAMES, the connection completes.

Once the connection or addresses are established, each machine must run the FTP program. Its prompt is FTP>. You can enter commands directly to FTP or use the pop-up menu.

The transfer may be performed with either machine taking the active role; the alternate scenarios are described below:

1. PERQ #1 takes the active role and PERQ #2 is passive. They'll follow this script:

   PERQ #2: Runs FTP and starts polling
   PERQ #1: Runs FTP and then issues the next command
   PERQ #1: PUT SourceFile[~][DestinationFile]
   Both PERQs: #!#!#!#!#!...#!#!
   (this appears on the screen as the file is being transferred and a "hand cursor" moves from top to bottom to show percentage complete. When the hand cursor reaches the bottom, the transfer

is complete.)

2. PERQ #2 has the active role:

```
PERQ #1: Runs FTP and starts polling
PERQ #2: Runs FTP and then issues the next command
PERQ #2: GET DestinationFile[~][SourceFile]
Both PERQs: #!#!#!#!...#!
            (same as above)
```

The passive PERQ polls while the active machine processes the transfer.

The valid commands for FTP are:

```
GET  SourceFile[~][DestinationFile]
PUT  SourceFile[~][DestinationFile]
MODE mode (can be PERQ, VAX-11, or PDP-11;
           default is PERQ)
BAUD baud rate (initially set to 9600)
DEVICE line (either RS232 or ETHERNET)
ADDRESS localname[~]remotename
```

You can specify HELP whenever FTP requests input.

You can call FTP with one command line. For example, you can specify:

```
FTP GET DestinationFile
```

Control is returned to the command interpreter Shell after that one command is executed. Additionally, you can specify the MODE, BAUD, and DEVICE commands as switches to the GET or PUT commands. For example:

```
FTP GET file/Mode=PDP-11/BAUD=1200/DEVICE=RS232
```

The FTP command also accepts a switch that permits you to specify whether or not the file to transfer is a text file; specify /TEXT for text files and /BINARY for other files.

Sometimes the BitPad and pen can cause problems with file transfers. If you encounter any problems, unplug the pen or move it away from the BitPad.

If an FTP transfer fails, the current transfer aborts and FTP waits for another command (or exits if invoked with a command line). However, if FTP was invoked from a command file and the transfer fails, it restarts the transfer and repeatedly tries again until it succeeds.

20. Help

The Help command is used to display helpful information about other commands and the PERQ operating system. When issued without an argument, a message describing the PERQ and the use of PopUp menus is displayed. When an argument is included, information about the program specified by the argument will be displayed. The HELP key or the Help command without an argument gets you to the help utility, if available. The help utility erases the screen and then prints a list of all commands for which help exists. Type one of these names (or use the pop-up menu to specify one). The specific help is then printed for that command.

21. Linker

The Linker takes compiler-generated .Seg files as its input and produces a runfile with a .Run extension. The runfile is created by linking together all of the separately compiled modules that make up a program. The command line takes the form:

```
LINK Source{,Source}[~runfile]{/switch}
```

Examples of valid command lines include

```
LINK Program
```
where Program.Seg is the output of a compilation

```
LINK Program1, Program2
```
where Program1.seg and Program2.seg are compiler output files. The output will be Program1.Run.

```
LINK Program1, Program2~Program0/VERBOSE
```
where Program1.Seg and Program2.Seg are compiled .Seg files and Program0.run is to be the runfile. The VERBOSE switch specifies that the name of each import module is to be listed.

Any number of source files can be listed separated by commas. The switches currently available are:

/MAP=name	-creates a map file. If you don't specify a name after the equal sign, the MAP file will have the same name as the runfile with a .Map extension. A map file contains a listing showing placement and size of the code and data segments of the program as well as other linkage details. Thus, map files are useful in debugging user programs.
/USER	-this is the default. It specifies that the program being linked is a user program. The opposite is /SYSTEM.
/VERBOSE	-lists the imports of each module as the module is being processed.
/SYSTEM	-specifies that this is a system program.
/VERSION=nn	-links the program with the version of the system specified by "nn". If the run file name ends in a number (e.g., "LOGIN.42"), then the Linker uses the number as the version number. This can be overridden by using /VERSION=nn switch.

A word about versions. Every system has a version number. When you create a new system, you also should use a new system version

number. You then have to relink all the runfiles. This insures that the programs execute correctly with the new system. In order to prevent confusion, we have put the system version number into the name of certain critical system runfiles: Link, Shell, Login, and System (e.g. Login.5.run). When you link a file, it looks up the system run file based on the version in use. If no version is specified, it uses the current system version (which is displayed at the top of the screen when in the Shell). If a version is specified, a system run file with that number is used to resolve references to system routines. If the /SYSTEM switch is used, however, no run file is looked for since a new system run file is created. The syntax for creating a system run file is:

```
LINK SYSTEM~SYSTEM.NewNumber/SYSTEM/VERSION=NewNumber
```

where the /VERSION switch is optional.

To create runfiles for a new system, you link using the VERSION switch after making the system runfile. It's done like this:

```
LINK FileSpecification/VERSION=NewNumber
```

22. Login

Login initiates a user's session at a PERQ. It is called automatically when the PERQ is booted or when a user logs off using the BYE command. It can also be called explicitly by a user.

At boot time, Login asks for the date, time, your name, and password. When you invoke Login, it asks for your name and password. Login prompts you with the format it wants these in.

It can be called explicitly with the command line:

```
LOGIN [UserName][/switch(es)]
```

and will prompt you with the information it wants.

Login searches the System.Users file for the given user-name and validates the password. The UserControl program (see below) maintains the System.Users file. If the user has a profile, it is read to find parameters to set up the PERQ for the user.

The Login command line may include switches. You can also include Login switches in a #Login section of your profile. The following describes the valid Login switches:

/PATH=pathname

Sets the default path to pathname. The /PATH switch is not cumulative, if you specify the switch multiple times, only the last one has an effect.

/SETSEARCH=pathname

Pushes pathname onto the search list. The effect of this switch is cumulative; if you specify the switch multiple times, you add additional items to the search list. If the argument is a minus sign (-), the last path is popped from the list. Note that the last item specified is the head of the searchlist and the first item specified is the end of the list.

/CURSORFUNCTION=n

Sets the default cursor function. Since the cursor function determines the screen color also, this switch can be used to set the default screen color.

Valid cursor functions are the integers 0 through 7 inclusive.
The integers signify the following:

0 - screen is all white

1 - only cursor displays
2 - white on black, cursor is large square
3 - black on white, cursor is large square
4 - black on white, black cursor hides image
5 - white on black, white cursor hides image
6 - black on white, cursor inverts
7 - white on black, cursor inverts

If you like white letters on a black background, use CURSORFUNCTION 5.

The default value is 4.

/SCREENBOTTOM=option

Sets the default parameters for the bottom of the screen when you change the screen size with a ScreenSize command (see below).

Valid options for this command are:

ON - displays bits stored in area
OFF - bottom is all one color
WHITE - bottom matches background
BLACK - bottom is opposite color of background

/COMMAND=string

Execute the specified string as the first Shell command after login and before accepting commands from the keyboard.

/SHELL=filename

Specify an alternate command interpreter to run instead of the Shell. This is an aid in debugging a new or user written command line interpreter.

/PROFILE=filename

Specifies a file (filename) to execute as the profile file.

/HELP

Displays a brief explanation of the Login Utility.

A sample profile for login might be:

```
#Login  /Path=Sys:User>MyName>
        /SetSearch=Sys:Boot>Library>
        /SetSearch=Sys:Part3>Alt7>
        /CursorFunction=4
```

Note that the command names in the profile can be abbreviated. Additional documentation for Login can be found in the section "Turning on the PERQ" in the PERQ Introductory User Manual.

23. MakeBoot

MakeBoot writes boot files onto hard disk and floppy disk. It is self-prompting. Its command line is:

MAKEBOOT [RunFile]

See the section "Booting the Machine" in the PERQ Introductory User Manual for more details on booting. The manual How to Make a New System gives complete description of the MakeBoot program.

24. MakeDir

MakeDir creates a new empty directory. Its command line is:

```
MAKEDIR [FileSpecification]
```

where FileSpecification is the name for the new directory. If you do not specify the name you will be prompted for it. All directories have a .Dr extension; MakeDir will provide this automatically if you do not specify it.

MakeDir will not accept as its parameter the name ROOT or the name of any extant file that has a .Dr extension; it will print an error message if you pass it such a name.

See the PERQ Introductory User Manual for more information on files and directories.

25. Mount

The Mount command is used to attach devices to the filesystem. Devices must be mounted before files on them can be accessed by the fileSystem. The two Mount commands are:

```
MOUNT H[ARDDISK]
```

and

```
MOUNT F[LOPPY]
```

Note that these are the forms to use no matter what the name devices were given when partitioned. The device that was booted from is mounted automatically by the system.

Notes on Mount and Dismount:

1. It is imperative that you dismount FileSystem (as opposed to FLOPPY) floppies before you remove them from the floppy drive. If you fail to do this, the next floppy put into the drive may be overwritten and the fileSystem floppy is also likely to be corrupted.

2. Floppies written with the FLOPPY program cannot be mounted or dismounted.

3. A device may be mounted more than once.

4. It's a bad idea to dismount the device you're running from. If you do, the system will not be able to find the Shell.

26. ODTPRQ

ODTPRQ is a simple debugger for microcode and new operating systems. ODTPRQ runs on a PERQ other than the one that is being debugged. It communicates through a Link board that is plugged into the I/O option slot in the card cage. ODTPRQ has an online help facility. The ODTPRQ command line is:

```
ODTPRQ [StateFileName]
```

where StateFileName is the name of a State File.

27. Partition

Partition creates new partitions on a device (such as hard disk or floppy). It can also be used to modify the names and sizes of existing partitions. Creating a new partition destroys all old data in the area where the partition is made. The device should first be formatted before using Partition. After formatting, Partition is the first program to run. Its command line is:

```
PARTITION
```

The program is self-prompting. Refer to "Partitions and the Partition Program" in PERQ File System Utilities Manual before using it.

28. Patch

Patch allows you to examine and modify the contents of a disk file. To run it, type:

```
PATCH [filename]
```

If you have not specified a filename or if the specified file does not exist, PATCH prompts for the filename. When it has a valid file, it prompts with:

```
Read Block [0]?
```

Type return to look at the block number in brackets or type a new block number. You can also type HELP for some online documentation that describes the commands you can use.

When you ask to read a block, Patch displays it byte by byte or word by word on your screen in 32-rows. You can reference each byte with the indices 0 to 511. Patch permits you to make temporary or permanent changes to your file.

To access all hard disk blocks, you can patch the SYS: file.

29. Path

Your "path" is the directory you are currently using. To find named files, the system will first look in this directory and then in directories specified in your search list. (See SetSearch below). New files are created in this directory if a different one is not specified. The Path command changes the current directory. The command line is:

PATH [pathname]

Path does not affect the search list. If Path is called without an argument, it prints the current path. A new path can then be typed or Carriage Return to exit without changing the path. The final ">" of a directory name is optional for the path command so "Path Foo" changes the path to the directory foo.Dr. Path will print an error message if you try to Path to a nonexistant directory. If you attempt to set a Path from which the Shell cannot be accessed (which can happen if you change the SearchList), Path requires confirmation.

30. Pause

Pause can be used to suspend the execution of a command file and wait for user confirmation before proceeding. It takes a message as a parameter, prints it on the screen, and then waits for a carriage return on the keyboard before continuing. This command can be given by the user, but it is most useful in command files when some user action is required before proceeding, e.g., changing floppies.

31. PERQ.Files

Part of the documentation for each PERQ operating system is a file called PERQ.Files which describes all the files in the system and states which part of the system they are in. The PERQ.Files program is used to list portions of the PERQ.Files text file and to make command files. Run the program by typing:

```
PERQ.Files
```

and type "Help" when you are prompted. The program types out a comprehensive description on how to use the program.

32. Print

Print sends files to a printer (the GPIB address for Print is one). If the file does not begin with a form-feed character, Print supplies it. Also, Print supplies a form-feed at the end of every printed file. Its command line is:

```
PRINT [Filename][,filename2,...filenamen][/switch(es)]
```

For printers connected to the RS232 port, Print requires Z80 PROMs version 8.5 (or higher) for proper operation. You can find these two PROMs on the "IO" board (red color-coded) labeled with the version number. For printers connected to the GPIB (IEEE-488) port, the PROM versions are not relevant. Contact Three Rivers field service to exchange earlier PROMs.

The Print command accepts the following switches:

/TALL	tall characters (six lines per inch)
/SHORT	short characters (eight lines per inch) /SHORT is the default
/WIDE	wide characters (10 characters per inch)
/NARROW	narrow characters (16.5 characters per inch) /NARROW is the default
/SHIFT=n	shifts the listing n spaces to the right /SHIFT=0 is the default
/COPIES=n	specifies the number of listings /COPIES=1 is the default
/START=n	specifies the page number of the document to begin printing /START=1 is the default
/STOP=n	specifies the last page number of the document to print /STOP=lastpagenum is the default
/BAUD=n	sets the baud rate for printing /BAUD=9600 is the default
/TABS=n	tab stops every n characters /TABS=8 is the default
/TITLE	prints a title line at the top of each page /TITLE is the default for files with other than a DOC extension
/NOTITLE	omits the title line /NOTITLE is the default for *.DOC files
/BREAK	places a blank page between each page of the listing
/NOBREAK	omits the blank page /NOBREAK is the default
/HP	initializes to use the Hewlett-Packard 7310A
/LINEPRINTER	initializes to use the TI 810 (or similar) printer
/DIABLO	initializes to use the Diablo 630 daisy printer
/PLAIN	no initialization; use with generic printers /PLAIN is the default
/HELP	supplies online documentation

33.PRQMic

PRQMic is the PERQ microcode assembler. It takes a microcode source program (whose extension is .MICRO), and produces output that can be used as input to the microplacer, PRQPlace. The PERQ Microprogrammers Guide has details on how to write microprograms and how to use PRQMic.

34.PRQPlace

PRQPlace is the PERQ microcode placer. It takes the output from the microassembler, PRQMic, and produces a file (with extension .BIN) that can be loaded into the PERQ microstore. The PERQ Microprogrammers Guide has details on the use of PRQPlace.

35. QDis

QDis is a disassembler for Q-Code. It decodes a .Seg file into Q-code. The command line:

QDIS

causes the following sequence:

1. The program identifies itself as QCode Disassembler.

2. You are then prompted for an input file. Type the name of the program or module you want disassembled. (Since the input to QDis is a .Seg file, the input file must have been compiled.) You needn't specify .Seg but must specify extension if it is anything else.

3. Next, you are prompted for an output file. The default is CONSOLE: .

4. QDis displays a list of the program's routines and the following information about each:

   Routine name
   Routine number
   Lexical level
   Parameter size
   Result + Parameter size
   Local + Temporary size
   Entry address
   Exit address

5. QDis next will ask you which routine you'd like to see disassembled. Type in a routine number; the program then types a listing of that routine's Q-code translation.

Step 5 can be repeated indefinitely.

36. Rename

Typing the command line:

RENAME SourceFile[~]DestinationFile

will change the name of SourceFile to DestinationFile. You can rename a file from one directory to another (move the file) as long as both directories are in the same partition.

Rename prompts for any missing arguments.

You may rename a .Run file, but if you rename a .Seg file you must re-link the programs which use it.

The source file for Rename may contain wild cards (for a description of the wild cards, see the "PERQ Introductory User Manual"). If the source contains wild cards, the destination must contain the same wild cards in the same order. In this case, Rename finds all files in the directory which match the source pattern. For these files, the part of the source file name that matched each wild card is used to replace the corresponding wild card in the destination. As an example, for the command:

RENAME foo*.abc# anotherdir>*baz.rmn#z

The input file "FOOZAP.ABCD" would be renamed to the new file "anotherdir>ZAPbaz.rmnDz".

If wild cards are used, Rename asks for verification of each file renamed. This can be disabled with the switch "/NOASK" or enabled with the switch "/ASK." The default is enabled.

Wild cards are not allowed in the directory part of the source file.

When the source file name contains no wild cards, the destination file name may contain, at most, one occurance of the wild card "*". In this case, the non-directory part of the source replaces the "*" in the destination. For example,

RENAME sys:Boot>newOS>myprog.Pas dir3>new.*

would rename the file "sys:Boot>newOS>myprog.Pas" to a new file named "dir3>new.myProg.Pas". This is most useful when you want to rename a file from one directory to another with the same name. For example,

RENAME dir1>prog.Pas *

moves prog.pas from the directory "dir1" into the current directory.

If there are no wild cards in the source, an attempt to include in the destination name other wild card characters, besides the single "*" discussed above, may lead to problems later. These extra wild

cards will be treated as simple literal characters. Because a file name with wild cards in it is hard to specify, Rename requires confirmation before creating a file with wild cards in the name.

If the the destination file already exists, Rename requests confirmation before deleting. This can be disabled by using the switch "/NOCONFIRM" or enabled using the switch "/CONFIRM". /NOCONFIRM also sets /NOASK. If an error is discovered and wild cards were used, Rename asks the user whether to continue processing the rest of the files that match the input. This confirmation is required no matter what switches were specified.

A final switch is "/HELP" which describes the function of Rename and the various switches.

37. ReRun

ReRun is a convenient method to reexecute the default file remembered by the Shell and supply new arguments to the runfile. The command line is:

```
ReRun arguments
```

The ReRun command is most useful when the default file has an especially long name. For example, if "Programwithlongname" is the default file, the command:

```
ReRun A b 3
```

is the same as typing:

```
Programwithlongname A b 3
```

38. Run

Run is another way to invoke the default file remembered by the Shell. If you have been editing, compiling and linking the default file, simply typing:

```
RUN
```

executes that default file. The Run command sets the default file to the specified runfile. For example, the command:

```
RUN Foo arg1 arg2
```

is the same as typing:

```
Foo arg1 arg2
```

except that the first command (Run Foo arg1 arg2) sets the default file to Foo.

39. ScreenSize

The display on the PERQ screen is stored in memory and requires 48K words (192 blocks). Sometimes it is advisable to give up some of the screen and allow this memory to be used for storing data or programs. For example, the Scavenger runs with swapping off so it shrinks the screen to allow its data and code to fit into memory. Even when swapping is enabled, some programs, such as the compiler, want to trade speed for screen size.

The ScreenSize command prompts for the number of scan lines used in the display for the next program run. The screen expands to full size after that program has completed. The full screen has 1024 lines in it. The number you supply must be greater than zero, less than or equal to 1024, and a multiple of 128 (e.g. 128, 512, 768, etc.). You can specify a number in the range 1 through 8 and ScreenSize will multiply the number by 128. For example, if you specify the value 4, ScreenSize multiplies 4 by 128 and uses 512 as the number of scan lines (512 is half of the screen).

ScreenSize accepts four switches to permit you to control the bottom portion of the display. The switches are:

/ON

This switch permits you to see the data or code that is stored in the screen area. The memory manager is told that the memory is free, but the IO package still thinks it should be displayed. The screen package will not let you create a window or write into that area (unless a program calls RasterOp or Line directly). This switch is the default condition, unless an entry in the user profile has changed it (see Login above).

/OFF

This switch forces the bottom of the screen area to a solid color.

/WHITE

This switch forces the bottom of the screen area to the same color as the usable part of the screen.

/BLACK

This switch forces the bottom of the screen area to the opposite color of the general background.

The Shell shrinks the screen automatically for certain programs. If

you explicitly call ScreenSize before these programs are run, then the Shell will not override your settings.

40. Scavenger

The Scavenger checks the filesystem's structures and fixes any errors found. It can be called any time you suspect a filesystem problem or whenever you need to reconstruct a directory. It is self-prompting.

Its command line is:

SCAVENGE

For more details on this program, see the section "Scavenger Program" in the PERQ File System Utilities Manual.

41. SetBaud

SetBaud allows you to specify the baud rate to the RS232 line. Valid baud rates are: 110; 150; 300; 600; 1200; 2400; 4800; and 9600. The command syntax is:

```
SetBaud 4800
```

42. SetSearch

SetSearch allows you to add and remove paths to search lists and to change their order. Its command line is:

```
SETSEARCH [pathname][,pathname]
```

If you do not specify a pathname, SetSearch displays the current search list and then permits you to change it or simply exit.

If you specify a file name, SetSearch pushes that name onto the search list; if it is a hyphen (-), the first item on the list is popped off.

The current search list is 5 items deep with the first and last slots reserved. SetSearch gives you a warning if you attempt to push something onto the first slot or pop something off the last. In addition, when you try to exit, SetSearch checks to make sure that the Shell can be found with the new search list and path. If not, you are asked for confirmation before exiting.

43. SetTime

The SetTime command allows you to specify the date and time. Specify the time and date as follows:

```
DD MMM YY  HH:MM:SS
```

The time notation uses a 24 hour clock and the seconds are optional. An example command line which sets the date to June 3, 1955 and the time to 3:30 PM folllows:

```
SetTime 3 Jun 55  15:30
```

SetTime permits you to change only the time; if you omit the date, you correct the time.

44. Statistics

The operating system constantly collects statistics about the performance of the swapper. To display the statistics after each execution completes, specify the command:

```
STATISTICS Yes
```

To end the display, type the command:

```
STATISTICS No
```

After each program is executed, the Shell prints the Statistics for that program in the following format:

```
Load      1.6 secs.
Exec      3.6 secs.
IO        0.8 secs.
Swap      0.1 secs.
Move      0.0 secs.
Duty     97.2 percent.
```

Load is the time spent loading the program and its modules into memory. Exec is the total time spent executing including IO, Swap and Move times. IO is the time spent doing Unit-level IO not including IO time spent swapping. Swap is the amount of time spent swapping. Move is the amount of time spent moving segments from one place in memory to another to try to find room for new allocation. Duty factor is the proportion of time spent in actual work. It is computed in the following way:

```
100 * (Exec - Swap - Move) / Exec
```

45. Swap

The Swap command enables or disables the virtual memory system. The command "SWAP NO" turns virtual memory off, and swaps all active segments into memory. "SWAP YES" turns swapping on. In this case, you may specify a partition to use for the swap files. For example,

```
SWAP YES Sys:Boot>
```

enables swapping to the Sys:Boot> partition. When booting from the hard disk, swapping is initially enabled in the partition containing the boot file. If you simply specify SWAP YES with no partition name, this default partition is used. When booting from the floppy, swapping is initially disabled. We discourage swapping to a floppy disk because floppies have a small capacity, are slow, and may be dismounted at any time.

46. TypeFile

Type displays any file or files on the console. The command line:

TYPEFILE FileSpecification

displays a single file on the console. To display multiple files, sequentially, separate the file specifications with a comma (,).

When the TypeFile command finds a formfeed character (^L) in the file or when it gets to the bottom of the screen, it waits after displaying a screenful of text. This permits the user to read the page before the screen is erased and the next page is displayed. To continue, type ^Q. This waiting can be disabled by using the /NOWAIT switch. The /WAIT switch, which is the default, causes the waiting to happen. The final switch available is /HELP, which displays online documentation.

The TypeFile command displays a solid, left pointing triangle when it reaches the end of the file.

TypeFile does extension completion on the file name specified. If the file to print is FOO.PAS, it is only necessary to type FOO. The extensions that type knows about, in order tried, are: Pas, Micro, Cmd, Dfs, and Doc.

If no filename is supplied on the command line, TypeFile will display the default file name. This is the same default file used by the Editor, Linker, Run, etc. Unlike these programs, however, TypeFile does not change the default file name.

47. UserControl

UserControl is used to maintain the System.Users file which contains information about valid users and their passwords, group identifiers, and user profiles. This information is used by the Login program (see above). The System.Users file must be in the root directory (top level) of the partition where the boot file is. Its command line takes the form:

USERCONTROL [command][/switch(es)]

If you include a command, UserControl executes the command and then exits to the Shell. If you do not include a command, UserControl prompts with:

UC>

and waits for input.

The following are valid commands:

HELP

Provides online documentation.

ADDUSER [username][/switch(es)]

Adds a new user to the user file or, updates the information for an existing user. If you omit a username, the command prompts for one.

Any printing character except blanks, spaces, equal signs (=), commas (,), or slashes (/) are valid usernames. The maximum number of characters is 31. The command accepts the following switches:

/PASS - permits an existing user to change his password or enters a password for a new user. Note that when you specify the /PASS switch, UserControl prompts for the password. You can enter any printing character except blanks, spaces, equal signs (=), commas (,), or slashes (/). The maximum number of characters is 31. You can also respond to the password prompt with CR. This causes a null password to be entered for the user.

If you omit the switch for a new user, the password defaults to null. For an existing user, the password is unchanged if the switch is omitted.

/GROUP=[group id] - permits an existing user to change the group id or enters a group id for a new user. The value for group id can be any integer from 0 to 255 inclusive. If you do not specify a group id, you are prompted for a value.

If you omit the switch for a new user, the group id defaults to 1. For an existing user, group id is unchanged if the switch is omitted.

/PROF=[profile] - specifies the complete path for the user's profile file.

If you omit the switch for a new user, the default is SYS:USER>name>PROFILE. For an existing user, the profile is unchanged if you omit the switch.

CHECKUSER [username]

Validates a username and password. If you omit the username, the command prompts for one.

NEWFILE

Destroys the existing System.Users file and creates a new System.Users file.

LISTUSERS

Displays a list of all of the valid users.

REMOVEUSER [username]

Delete a user from the file. If you omit the username, the command prompts for one.

QUIT

Exits the program.

You can also specify the /HELP switch (type /HELP or press the HELP key) with any command or in response to any prompt to display specific help information.

Editor V2.0 User's Guide

John P. Strait
W. J. Hansen

This manual describes version V2.0 of the PERQ text editor. It includes a small amount of overview and philosophy behind the Editor and contains a list of the commands available. The reader is expected to have a general grasp of computers and computerized editors. Some experience or familiarity with a similar editor is helpful.

Changes

The major feature introduced with this version of the editor is a set of commands to control the selection from the keyboard. A few features of the editor have been modified:

- Find can ignore the case of the text.
- Find and Replace can be interrupted with control-C.
- The reverse direction flag (<) is turned off by most commands.
- The verify flag (V for Replace Command) is turned off by most commands.
- The replot function is invoked by Q,R,Return (not X).

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

Table of Contents

Introduction

It is probably a good idea to sit down at a PERQ and try out the Editor as you are reading. A good file to edit when you are first trying out the Editor is one of the Editor's help file. This way you can read the help file while you are editing. You should copy it to a scratch file so that you won't have to worry about accidentally changing it. Do this with the system "Copy" command:

```
Copy >HelpDir>EditorHelp>Introduction.Help Scratch.File
```

Now ask the Editor to edit this new scratch file:

```
Edit Scratch.File
```

The PERQ text Editor is a "point, act" Editor. This means that to perform an editing action, you first "point" to a piece of text in the file and then perform some action on it. Pointing is done with the tablet and puck or pen. When you move the puck on the tablet, the pointer on the screen moves to follow it. The pointer changes shape depending on where it is on the screen. It is usually an up-and-left pointing arrow, but changes in certain areas to an up-pointing arrow, a down-pointing arrow, or a circle. These different shapes indicate that different things will happen when you press the buttons on the puck (or press down with the pen).

To select a specific piece of text, move the puck so the pointer is at the text and press the yellow button. With one press a character is selected. A second press without moving the pointer selects the word of adjacent letters and digits. A third press selects the entire line. Another way to select an entire line is to press when the pointer is at the left margin line. The other buttons on the puck can also be used to make selections: the white button selects a word, the green button selects a line, and the blue button extends the selection to the location pointed at.

Once you have selected some text, you can extend the selection by moving the pointer to the end of the desired piece of text and then typing "E". The selection is extended to a character, word, or line boundary, depending on the type of your last selection. Commands that change the selection (other than Extend, W, and L) set the type to character. Unfortunately, you can make the selection larger with Extend, but you cannot make it smaller. To do so, you can start over with a fresh selection or use the M command described below.

After selecting text, you can perform some action on it. You can insert before or after it, delete it, search for a character string that follows it, or any of the other Editor actions. The top line of the screen usually shows editor version, file name, and time. It also displays repeat counts and flags for More, Verify, and direction. On occasion the line is used for error messages and prompts within commands.

When the first line of text is showing on the screen, it is indicated by a "▶" in the left margin--this is the beginning-of-text marker. When the last character of the file is showing on the screen, it is followed by a "◀"--this is the end-of-text marker. You may select this character, but it is not affected by text modification commands.

The following sections describe the commands available. Each command description begins with the letter on the key used to perform the command.

First Things First

HELP - Get explanations

Type the HELP key to get explanations of the editor commands. (If you are "in" a command, you may need to type DEL before typing HELP.) The editor will display a list of commands and keywords. You may then type the name of an entry (the word or letter before the dash) and the editor will display an explanation. To exit, just type the RETURN key.

Q - Quit from the Editor

To leave the editor, type the "Q" key. (If you are "in" a command, you may need to type DEL before typing "Q".) When you type "Q", the screen is erased, and a list of options is presented:

U to update <your file name>
W to write to another file
E to exit without updating
R to return to the Editor

Type one of these letters followed by RETURN. You may wish to pause before the RETURN to be sure you have the right choice.

If you update or write to a file that already exists, the Editor saves the old version of the file by adding a "$" to the end of its name. This allows you to get the old version back if you decide you made a mistake. You can safely edit the backup file since it has a different name than the new version of the file.

While it is never a good idea to type control-C while you're in the Editor, you should not type it while the Editor is writing the new copy of your file. You will lose the new version of your file, and the old version will be in the backup file. If you type control-C before writing the new file or type "E" to exit without updating, your file will remain unchanged regardless of any changes you made with the Editor.

Special Keys

The Text Modification commands are terminated by "Accept" or "Reject". The first of these indicates that the change is desired, the editor makes the change. The second foregoes the change, the editor leaves the text as it was before the command. Acceptance is signalled by typing the INS key. Rejection is signalled by typing the DEL key.

When inserting text with the Editor, BACK SPACE deletes the most recently typed character, control-BACK SPACE deletes the most recently typed word, and control-OOPS deletes the most recently typed line up to and including the carriage return. The RETURN key (carriage return) is used to mark the end of each line in your file. The Editor does automatic indenting for you by supplying leading blanks on the new line to match those on the previous line. You may BACK SPACE over them or type more.

The special keys have alternates as shown by this table:

Action	Character	Alternate
End-of-line	RETURN	control-M,control-J -or- control-M,LF
Erase character	BACK SPACE	control-H
Erase word	control-BACK SPACE	control-W
Erase line	control-OOPS	control-U
Accept	INS	control-Y
Reject	DEL	control-N
Quote	control-"	-none-

Since these special keys have special meanings, they must be "quoted" to insert them in your file; Type control-" (the Editor quote character) followed by the special key. When you type control-", the insert cursor changes from "_" to "□" to indicate you typed the quote character.

INS - Repeat the last command

Typing the INS key at command level repeats the last command. Only certain commands may be repeated this way, since it does not make sense to repeat some commands like * and Extend. Commands that may be repeated are A, I, R, S, D, and those which move the selection.

Repeat Count

Typing a 1- to 4-digit number enters a repeat count; it is displayed in the prompt line after the letter "R". This number specifies how many times to execute the next command, but it only applies if the command is Find, Repeat, Goto character, LF(Scroll), Line, Word, or one of the Character Selection commands. The number of repetitions actually performed is displayed in the prompt line after the letter "C".

Reverse Direction

Many commands move through the file. They can do this either forward toward the end of the file or in reverse toward the beginning of the file. The current direction is displayed by a "<" or ">" in the upper left corner of the screen. It effects the commands Find, Replace, Word, Line, and Goto character. The flag is turned forward by all other commands except More, Extend, Verify, and Note. It is also turned forward by Find, but the old direction is remembered if the Find is repeated with INS.

> - Forward Direction
Typing a ">" sets the forward direction (toward the end of the file). "+" and "." (unshifted ">") are synonyms for this command.

< - Reverse Direction
Typing a "<" sets the reverse direction (toward the beginning of the file). "-" and "," (unshifted "<") are synonyms for this command.

View Selection

The Scroll Bar

The area to the left of the left-margin line is called the "scroll bar". If you move the pointer into this bar, it changes into an up-pointing arrow at the left side and a down-pointing arrow at the right side. If you press the yellow button when the arrow is pointing up, the line that the pointer is next to is scrolled to the top of the screen. If you press when the arrow is pointing down, the top line of the screen is scrolled down to the line that the pointer is next to. Thus if you put the pointer near the top of the screen, by pressing repeatedly, you can scroll slowly through the file. If you put the pointer near the bottom of the screen, you can move through the file in large jumps of pages. If you put the pointer near the middle of the screen, you can move through the file

in half-page jumps.

When the pointer is in the Scroll Bar area, the puck buttons have special meanings: the white button always scrolls the text up and the green button always scrolls it down.

LF, Control-LF - Scroll
LF repositions the screen window so it displays text starting with the 37th line of the previous display. Control-LF scrolls in the other direction so the previous top of the screen is at the new bottom.

T - Top
The line containing the end of the current selection is moved to the screen. If on-screen, it is moved to the bottom.

B - Bottom
The line containing the current selection is moved to the bottom of the screen. (If off-screen, it is moved to the middle.)

The Thumb Bar

The top-margin line is called the "thumb bar". When you move the pointer to this line, it changes into a circle. Think of the thumb bar as a linear representation of your file. The left end of the bar represents the beginning of your file. Special characters are used to represent other interesting parts of your file:

◀ - Represents the end of the file.

S - Represents the position of the beginning of the selection.

(- Represents the position of the beginning of the displayed text.

) - Represents the position of the end of the displayed text.

N - Represents the position of the Noted display.

O - Represents the position of the display at the last thumbing.

The thumb bar is used to rapidly move around in your file, but it is not very precise. When you put the pointer on the thumb bar and press down, the portion of the file represented by that portion of the bar is displayed on the screen. When you press at "S", the beginning of the selection is shown. For "N", the noted position is shown. And so on. Similar control can be achieved with the X

command described below.

The thumb bar can also be used to extend the selection. When you type "E" while in the thumb bar, the selection is extended to the position in the file which is represented by that particular portion of the thumb bar. This is usually only useful for extending to the beginning or end of the file.

N - Note
The current screen display is noted and an N is placed in the Thumb Bar. The thumb bar may be used to return to this current display by selecting the N.

Text Selection

E - Extend the selection
This command extends the current selection from where it is to the text currently indicated by the pointer. See the M command for an alternative way to extend the selection.

* - Select the entire file
The entire file is selected. This is useful for Find, which otherwise begins its search from the current selection. It is also useful for doing replacements throughout the text.

F - Find a character string
(N. B.: The search starts at the beginning of the selected text.) When you type "F", the top line shows the prompt

Find: enter target string

Type the string you want to search for and then Accept or Reject. Rejection aborts the Find command; Acceptance starts the search. If you Accept immediately after typing "F", the previous target string is used. Lower case letters in the target string will match both upper and lower case letters in the text, but UPPER case target letters will match only upper case text letters.

While searching for the target, the top line displays the cue "Finding". At this time, the search can be interrupted by typing control-c. The selection will be unchanged.

If the target string is found, it becomes the selected text. Note that the target string is shown at the top of the screen inside of

F{ ... } .

A Find command can be done in the reverse direction (toward the beginning of the file). See the section "Reverse Direction". You can ask the Editor to search for a certain number of occurrences of the target string by preceding the command with a number. See the section "Repeat Count".

Text Modification

Text modification is done with the commands Insert, Append, Substitute, Replace, and Delete. The first two of these operate before and after the current selection, but the others operate on the current selection itself. After execution of one of these commands, it can be re-executed by pressing the INS key. This can be handy for inserting the same text in a number of places.

A - Append text after the selection. Type text until you are done, then either Accept or Reject the insertion. If you Accept immediately after typing "A", the most recently inserted or deleted text is inserted. If you Accept, the text you have just typed is displayed at the top of the screen inside of

I{ ... } .

D - Delete the selected text. If you delete text, it is displayed at the top of the screen inside of

D{ ... } .

The character which immediately follows the deleted text is selected.

I - Insert text before the selection. Type text until you are done, then either Accept or Reject the insertion. If you Accept immediately after typing "I", the most recently inserted or deleted text is inserted. If you Accept, the text you have just typed is displayed at the top of the screen inside of

I{ ... } .

R - Replace
This command finds all occurrences of a given string within the currently selected text. As each is found, it is replaced by a second given string. Typing "R" causes the prompt

Replace: enter target string

at the top of the screen. Type in a string and then Accept or Reject. Note that the target string is displayed inside of

F{ ... }

just as though you were using the the Find command. Lower case letters in the target string will match both upper and lower case letters in the text, but UPPER case target letters will match only upper case text letters. If you Accept, the prompt

Replace: enter replacement string

is shown at the top of the screen. Now you can type in the new string to replace occurrences of the target string. If you Accept before typing any characters of the replacement string, the previous replacement string is used. This means that to replace with nothing, you must first type a character and then delete it with the BACK SPACE key.

While the editor is "Replacing", you may interrupt by typing control-c. The Count field at the top of the screen will show how many replacements have been made.

In the absence of a Repeat Count, all occurrences within the selection are replaced. If a Repeat Count is typed immediately before the "R", the specified number of occurrences are replaced, starting at the beginning of the selection. See the section "Repeat Count". The direction can be changed as described in the section "Reverse Direction".

V - Toggle Verify mode

When you type "V", Verify mode is turned on or off, depending on whether it was off or on before. When Verify mode is on, a "V" is displayed in the prompt line. Most commands revert the verify flag to non-verify mode. Those that do not cause reversion are Word, Line, *, Goto character, More, Extend, Note, and the Character Selection commands.
In Verify mode the Replace command gives you the option of replacing, not replacing, or aborting at each occurrence of the target string. The prompt

Replace: INS replaces, <space> doesn't, DEL aborts

is displayed for each occurrence of the target string. The target string is indicated by a double underline and by the pointer arrow (if you keep the puck or pen away from the tablet). This double underline is difficult to see, but it's there. You may now Accept the replacement with INS, you may skip over this occurrence by typing the space-bar, or abort the replace command with DEL.

S - Substitute
New text is substituted for the selected text. This command is similar to the sequence "Delete, Insert". After typing "I", type text until you are done, then either Accept or Reject the insertion. If you Accept immediately after typing "S", the most recently inserted or deleted text is inserted. If you Accept, the text you have just typed is displayed at the top of the screen as the most recently inserted text inside of

I{ ... }

and the text you have just deleted is displayed at the top of the screen inside of

D{ ... } .

After substituting, the character immediately following the deleted text is selected.

Be careful. It is easy to confuse Substitute and Replace. There is no way to "undo" a Substitute command because you cannot re-insert the deleted text. Attempting to re-insert the most recently inserted or deleted text will merely re-insert the text you just typed in. The deleted text is gone forever. This means that if you type "S" when you meant to type "R", you may accidentally delete a large portion of your file and have no way of getting it back. (Should disaster strike, see the section below on "Transcript/Replay".)

Character Selection

The Character Selection commands move the selection as though it were a cursor for insert. The new selection is a single character close to the former selection. None of these commands is affected by the Reverse Direction flag, but all are performed as many times as specified by the Repeat Count. They reset the Reverse Direction flag unless the More flag is set. When the More flag is set, these commands still move in their usual direction; thus they can be used to reduce the selection.

SPACE - Move to the character following the previous selection.

BACKSPACE - Go to the character preceding the previous selection. Control-H is synonymous with this command.

TAB - Advance five characters from end of previous selection.

Control-TAB - Go back five characters from beginning of previous selection.

Control BACKSPACE - Go to the beginning of the word preceding the current selection. Control-w and Control-W are synonymous with this command, but shift-control-W treats a word as any consecutive string of printable characters.

RETURN - Move to first character of line following the previous selection. Control-U is synonymous with this command.

Control-RETURN - Go back to first preceding non-blank that follows a CRLF. Control-OOPS is synonymous to this command.

X - Thumb bar (X-Coordinate selection)
An "X" is placed on the thumb bar at the top of the screen. The following keys are active:

INS - Display the part of file corresponding to current position of the "X".
DEL - Aborts. Display and current selection are unaffected.
TAB - Move "X" five positions to the right.
control-TAB - Move "X" five positions to the left.
SPACE - Move "X" one position to the right.
BACKSPACE - Move "X" one position to the left.
RETURN - Move "X" to end of file position.
Control-RETURN - Move "X" to beginning of file position.
Digits - Set repeat count for TAB's and SPACE's.
O - Move to O marker on line. This marker indicates what text was displayed prior to the last thumb bar selection.
N - Move to the N marker on the line. This marker is set by the N command.

The current selection is unchanged by X. However, if a keyboard command (C, W, L, G, space, ...) is given when the selection is off screen, the operation is treated as though the first character of the screen had been previously selected.

G - Go to character

After typing "G", one more character is typed. The editor searches the rest of the screen for the character. If found, it becomes the current selection. If the Reverse Direction flag is set, G searches from the current selection toward the top of the screen otherwise it searches only from the selection downward. INS will repeat a G command if it was the last command exectued. A repeat count may also be given to select a character a known number of instances away. If the More Flag is set, the selection is extended to the found character.

w, W - Select Word

This command selects the word that follows, extends, or begins the current selection. If the current selection is a word or is outside a word, the following word is selected; if currently inside a word, the entire word is selected. When the current selection overlaps more than one word, the one overlapping the beginning of the selection is chosen. Unshift-w chooses words that are consecutive letters and digits; Shift-W defines a word as any sequence of printable characters.

A Repeat Count of (say) n will cause the selection of the n'th subsequent word. If the Reverse Direction flag is set, the search will be toward the beginning of the file. If the More flag is set, the selection is extended to the end of the word that would otherwise be selected by this command.

L - Select Line

If a line is currently selected, this command selects the next; otherwise it extends or contracts the current selection to be the line that included its start.

A Repeat Count of (say) n will cause the selection of the n'th subsequent line. If the Reverse Direction flag is set, the search will be toward the beginning of the file. If the More flag is set, the selection is extended to the end of the line that would otherwise be selected by this command.

M - More in selection

Turns on the More flag, indicated by an M in the prompt line. When this flag is on, selection commands extend the current selection rather than change it. The selection grows only at the end indicated by the Reverse Direction flag. The Character Selection commands (SPACE, BACKSPACE, RETURN, etc.) can cause the selection to shrink at that end. In More mode, these commands do not select a single character, nor do they change the direction flag. More mode is extinguished by More again or by executing *, Extend, Quit, or one of the Text Modification commands.

Transcript/Replay

The Editor writes a transcript file during every edit session. The transcript is a file which contains a description of every keystroke and puck or pen press performed during an edit session. This transcript is written to the file ">Editor.Transcript". The transcript may be replayed later. This feature is intended for use when the Editor or the PERQ crashes during an edit session or if you make some disasterous error with the Editor.

The Editor saves keystrokes and presses and writes them to the transcript file whenever:

1) A carriage-return is typed in Insert mode.
2) A command which changes the text is successfully completed.
3) The transcript buffer is filled.

If an old transcript file exists, it is destroyed the first time the buffer is flushed. This means that you can re-enter the Editor without destroying the old transcript file as long as you do not do something that causes the buffer to be flushed. Keep in mind that presses count against the 256-word buffer. If you do not want to destroy the old transcript file, do not type any commands, do not press more than a few times, and exit the Editor by typing control-shift-C.

To replay a transcript, type "Editor/Replay". The Editor replays the previous edit session and stops just before the first command. You can control the replay by typing one of the following keys:

SPACE	stop replaying after the next character or puck press.
CR	stop replaying after a carriage return in I command or after next command if not in I command.
LF	stop replaying after next command.
INS	begin replaying and stop when one of the above keys is typed.
DEL	exit replay mode.

If no DEL key is typed, the Editor automatically exits from replay mode when the end of the transcript is reached. Once you have exited replay mode you can begin editing normally, but we suggest that you Quit-Update as soon as possible.

PERQ Pascal Extensions

Miles Barel
Michael Kristofic

January 7, 1982

PERQ Pascal is an upward-compatible extension of the standard programming language Pascal. This document describes only the extensions to PASCAL.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

Table of Contents

Table of Contents

1. Introduction

PERQ Pascal is an upward-compatible extension of the programming language Pascal defined in PASCAL User Manual and Report [JW74]. This document describes only the extensions to Pascal. Refer to PASCAL User Manual and Report for a fundamental definition of Pascal. This document uses the BNF notation used in PASCAL User Manual and Report. The existing BNF is not repeated but is used in the syntax definition of the extensions. The semantics are defined informally.

These extensions are designed to support the construction of large systems programs. A major attempt has been made to keep the goals of Pascal intact. In particular, attention is directed at simplicity, efficient run-time implementation, efficient compilation, language security, upward-compatibility, and compile-time checking.

These extensions to the language are derived from the BSI/ISO Pascal Standard [BSI79], the UCSD Workshop on Systems Programming Extensions to the Pascal Language [UCSD79] and, most notably, Pascal* [P*].

2. Declarations

You must declare all data items in a Pascal program. To declare a data item, specify the identifier and then what it represents.

2.1 Identifiers

PERQ Pascal permits the inclusion of the underscore character "_" as a significant character in identifiers.

2.2 Declaration Relaxation

The order of declaration for labels, constants, types, variables, procedures and functions has been relaxed. These declaration sections may occur in any order and any number of times. It is required that an identifier be declared before it is used. Two exceptions exist to this rule:

1) Pointer types may be forward referenced as long as the declaration occurs within the same type-definition-part, and

2) Procedures and functions may be predeclared with a forward declaration.

The new syntax for the declaration section is:

```
<block> ::= <declaration part><statement part>

<declaration part> ::= <declaration> |
      <declaration><declaration part>

<declaration> ::= <empty> |
        <import declaration part> |
        <label declaration part> |
        <constant definition part> |
        <type definition part> |
        <variable declaration part> |
        <procedure and function declaration part>
```

Note: See "IMPORTS Declaration" in this document.

2.3 Files

PERQ Pascal permits the use of files as the component type of arrays, pointers and record fields.

3. Numbers

3.1 Whole Numbers

Single precision whole numbers are of the predefined type INTEGER. They occupy 16 bits (15 bits and a sign bit) and range in value from -32768 to +32767. Whole numbers of the predefined type LONG are double precision. They occupy 32 bits (31 bits and a sign bit) and range in value from -2147483648 to +2147483647. Arithmetic operations and comparisons are defined for both single and double precision whole numbers. See the section "Extended Constants" for a discussion of single and double precision constants.

3.2 Floating Point Numbers

PERQ Pascal floating point numbers (type REAL) occupy 32 bits and conform to the IEEE floating point format. See [FP80]. Positive values range from approximately 1.1754945e-38 to 3.402823e+38 and negative values range from approximately -1.1754945e-38 to -3.402823e+38 . Arithmetic operations and comparisons are defined for floating point numbers.

3.3 Type Coercion Intrinsics

The STRETCH intrinsic can be used to explicitly convert a single precision whole number into a double precision whole number. The SHRINK intrinsic will convert double precision into single precision, provided that the value of the double precision number is within the legal range of single precision. If it is not, a runtime error will occur. The FLOAT intrinsic will convert a single precision whole number into a floating point number. TRUNC and ROUND will convert a floating point number into a single precision whole number, provided that the value of the floating point number is within the legal range of single precision. If not, a runtime error occurs. There is no way to convert a floating point number into a double precision whole number or vice versa.

For example:

```
VAR R : real;
    L : long;
    I : integer;

    .
    .
    .

R := FLOAT(I);
I := TRUNC(R);
I := ROUND(R);
L := STRETCH(I);
I := SHRINK(L);

    .
    .
    .
```

3.4 Assignment Compatibility

Single precision whole numbers are assignment compatible with double precision whole numbers and floating point numbers. That is, expressions of type INTEGER can be assigned to variables of type LONG or REAL and INTEGER expressions can be passed by value (only) to LONG or REAL formal parameters. Double precision whole numbers and floating point numbers are not assignment compatible with single precision whole numbers. The type coercion intrinsics SHRINK, FLOAT and TRUNC can be used for this purpose. Double precision whole numbers and floating point numbers are not compatible in any way whatsoever.

3.5 Mixed Mode Expressions

Mixed mode expressions between single and double precision whole numbers or between single precision whole numbers and floating point numbers are allowed. Mixed mode expressions containing double precision whole numbers and floating point numbers are not allowed. IMPORTANT NOTE: A mixed mode expression is evaluated from left to right (taking normal operator precedence and parentheses into account) in single precision mode until the first LONG (or REAL) is encountered. Starting at that point, all single precision operands are converted to double precision (or floating point) before used in evaluation. Single precision overflow or underflow can occur while in single precision mode. The STRETCH (or FLOAT) intrinsics may be used to avoid this problem.

4. Extended Constants

4.1 Whole Number Constants

Unsigned octal whole number (INTEGER or LONG) constants are supported as are decimal constants. Octal constants are indicated by a '#' preceding the number.

The syntax for an unsigned integer is:

<unsigned integer> ::= <unsigned decimal integer> |
<unsigned octal integer>

<unsigned decimal integer> ::= <digit>{<digit>}

<unsigned octal integer> ::= #<ogit>{<ogit>}

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ogit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

NOTE: Unsigned constants do not imply unsigned arithmetic. For example, #177777 has the value -1, not +65535.

4.2 Single and Double Precision Constants

A constant in the range -32768 to +32767 (or in the range -#177777 to +177777, if expressed in octal) is considered single precision (an INTEGER). Constants exceeding this range are considered double precision (LONG). The maximums for double precision constants are -2147483648 (or -#37777777777) and +2147483647 (or +#37777777777).

4.3 Constant Expressions

PERQ Pascal extends the definition of a constant to include expressions which may be evaluated at compile-time. Constant expressions support the use of the arithmetic operators +, -, *, DIV, MOD and /, the logical operators AND, OR and NOT, the type coercion functions CHR, ORD and RECAST, the WORDSIZE intrinsic, and previously defined constants. (See "WordSize" in this manual and RECAST under "Type Compatibility", also in this manual.) All logical operations are performed as full 16-bit operations.

The new syntax for constants is:

```
<constant> ::= <string constant> | <constant expression>

<constant expression> ::= <csimple constant expression> |
        <constant expression>  <relational operator>
         <csimple constant expression>

<relational operator> ::= = | <> | < | <= | > | >=

<simple constant expression> ::= <cterm> | <sign><cterm> |
      <simple constant expression><adding operator><cterm>

<adding operator> ::= + | - | OR

<sign> ::= + | -

<cterm> ::= <cfactor> |
        <cterm><multiplying operator><cfactor>

<multiplying operator> ::= * | / | DIV | MOD | AND

<cfactor> ::= <unsigned constant> |
        ( <constant expression> ) | NOT <cfactor> |
        CHR(<constant expression>) |
        ORD(<constant expression>) |
        RECAST(<constant expression>,<type identifier>) |
        WORDSIZE(<name>)
```

5. Type Compatibility

PERQ Pascal now supports strict type compatability as defined by the BSI/ISO Pascal Standard [BSI79], with one addition; Any two strings, regardless of their maximum static lengths, are considered compatable.

5.1 Type Coercion - RECAST

The function RECAST converts the type of an expression from one type to another type when both types require the same amount of space. RECAST, like the standard functions CHR and ORD, is processed at compile-time and thus does not incur run-time overhead. The function takes two parameters: the expression and the type name for the result. Its declaration is:

```
function RECAST(value:expression_type; T:result_type_name):T;
```

The following is an example of its use:

```
program RecastDemo;
type color = (red, blue, yellow, green);
var  C: color; I: integer;
begin
I := 0;
C := RECAST(I, color);   { C := red; }
end.
```

Note that RECAST does not work correctly for all combinations of types; use the RECAST function sparingly and always scrutinize the results. Generally, only the following conversions produce expected results:

- longs, reals, and pointers to any other type
- arrays and records to either arrays or records
- sets of 0..n to any other type
- constants to long or real
- one word types (except sets) to one word types (except sets)

Avoid the use of the RECAST function for any other conversion.

WARNING: successful compilation does NOT imply that the RECAST function will execute correctly.

6. Extended Case Statement

Two extensions have been made to the case statement:

1) Constant subranges as labels.

2) The "otherwise" clause which is executed if the case selector expression fails to match any case label.

Case labels may not overlap. A compile-time error will occur if any label has multiple definitions.

The extended syntax for the case statement is:

```
<case statement> ::= CASE <expression> OF
        <case list element> {;<case list element>} END

<case list element> ::= <case label list> : <statement> |
        <empty>

<case label list> ::= <case label> {,<case label>}

<case label> ::= <constant> [..<constant>] | OTHERWISE
```

If the selector expression is not in the list of case labels and no OTHERWISE label is used, then the case statement is a no-op. This is different from PASCAL as defined by the PASCAL User Manual and Report, which suggests that this be a fault.

7. Control Structures

7.1 EXIT Statement

The procedure EXIT is provided to allow forced termination of procedures or functions. The EXIT statement may be used to exit from the current procedure or function or from any of its parents. The procedure takes one parameter: the name of the procedure or function to exit from. Note that the use of an EXIT statement to return from a function can result in the function returning undefined values if no assignment to the function identifier is made prior to the execution of the EXIT statement. Below is an example use of the EXIT statement:

```
program ExitExample(input,output);
var Str: string;

   procedure P;
   begin
   readln(Str);
   writeln(Str);
   if Str = "This is the first line" then
       exit(ExitExample)
   end;

begin
P;
while Str <> "Last Line" do
    begin
    readln(Str);
    writeln(Str)
    end
end.
```

If the above program is supplied with the following input:

```
This is the first line
This is another line
Last Line
```

the following output would result:

```
This is the first line
```

If the procedure or function to be exited has been called recursively, then the most recent invocation of that procedure will be exited.

WARNING: The parameter to EXIT can be any procedure or function name.

If the specified routine is not on the call stack, the PERQ will crash.

8. Sets

PERQ Pascal supports all of the constructs defined for sets in Chapter 8 of PASCAL User Manual and Report [JW74]. Space is allocated for sets in a bit-wise fashion -- at most 255 words for a maximum set size of 4,080 elements. If the base type of a set is a subrange of integer, that subrange must be within the range 0..4079, inclusively. If the base type of a set is a subrange of an enumerated type, the cardinal number of the largest element of the set must not exceed 4,079.

9. Record Comparisons

PERQ Pascal supports comparison of records with one restriction: No portion of the records can be packed. For example,

```
program P(input,output);
var
    R1,R2 : record
                RealPart : integer;
                Imagine  : integer;
            end;
begin
R1.RealPart := 1;    R1.Imagine := 2;
R2.RealPart := 1;    R2.Imagine := 2;
if R1 = R2 then writeln('Records equal');
end.
```

will produce as output

```
'Records equal'
```

10. Strings

PERQ Pascal includes a string facility which provides variable length strings with a maximum size limit imposed upon each string. The default maximum length of a STRING variable is 80 characters. This may be overridden in the declaration of a STRING variable by appending the desired maximum length (must be a compile-time constant) within square brackets after the reserved type identifier STRING. There is an absolute maximum of 255 characters for all strings. The following are example declarations of STRING variables:

```
Line : STRING;          { defaults to a maximum length of 80
                          characters }

ShortStr : STRING[12];          { maximum length of ShortStr
                                  is 12 characters }
```

Assignments to string variables may be performed using the assignment statement or by means of a READ statement. Assignment of one STRING variable to another may be performed as long as the dynamic length of the source is within the range of the maximum length of the destination -- the maximum length of the two strings need not be the same.

The individual characters within a STRING may be selectively read and written. The characters are indexed starting from 1 through the dynamic length of the string. For example:

```
program StrExample(input,output);
var Line : string[25];
      Ch : char;

begin
Line:='this is an example.';
Line[1]:='T';    { Line now begins with upper case T }
Ch:=Line[5];     { Ch now contains a space }
end.
```

A STRING variable may not be indexed beyond its dynamic length. The following instructions, if placed in the above program, would produce an "invalid index" run-time error:

```
Line:='12345';
Ch:=Line[6];
```

STRING variables (and constants) may be compared regardless of their dynamic and maximum lengths. The resulting comparison is lexicographical according to the ASCII character set. The full 8 bits are compared; hence, the ASCII Parity Bit (bit 7) is significant for lexical comparisons.

A STRING variable, with maximum length N, can be conceived as having the following internal form:

```
packed record DynLength : 0..255;       { the dynamic length }
              Chrs: packed array [1..N] of char;
         end;                 { the actual characters go here }
```

10.1 LENGTH Function

The predefined integer function LENGTH is provided to return the dynamic length of a string. For example:

```
program LenExample(input,output);
var Line:string;
    Len:integer;
begin
Line:='This is a string with 35 characters';
Len:=length(Line)
end.
```

will assign the value 35 into Len.

11. Generic Types

Generic types are general forms of more specific types. PERQ Pascal provides two predefined generic types:

1) Pointers

2) Files

11.1 Generic Pointers

Generic pointers provide a tool for generalized pointer handling. Variables of type POINTER can be used in the same manner as any other pointer variable with the following exceptions:

1) Since there is no notion of type associated with the reference variable of a generic pointer, generic pointers cannot be dereferenced.

2) Generic pointers cannot be used as an argument to NEW or DISPOSE.

3) Any pointer type can be passed to a generic pointer parameter. To make use of a generic pointer, RECAST should be used to convert the pointer to some usable pointer type.

The following is a sample program utilizing generic pointers:

```
program P(input,output);
type
    PtrInt = ^integer;
    PAOfChar = packed array[1..2] of char;
    PtrPAOfChar = ^PAOfChar;
var
    I : PtrInt;
    C : PtrPAOfChar;

    procedure Proc1(GenPtr : pointer);
    var W : PtrPAOfChar;
    begin
        W := recast(GenPtr,PtrPAOfChar);
        writeln(W^[1],W^[2])
    end;

begin
new(I);
I^ := 16961;    { First byte = 'A', second = 'B' }
Proc1(I);
new(C);
C^[1] := 'C';
C^[2] := 'D';
Proc1(C);
end.
```

will produce output

```
AB
CD
```

11.2 Generic Files

Generic files have very restricted usage. Their purpose is to provide a facility for passing various types of files to a single procedure or function. Generic files may only appear as routine VAR parameters. Their type is FILE. They can be used in only two ways:

1) Passed as a parameter to another generic file parameter,or

2) As an argument to the LOADADR intrinsic.

The following examples show two ways of using generic files:

```
program P;
type
    FOfInt = file of integer;
var
    F : text;
    F2 : file of boolean;

    procedure Proc1(var GenFile : file);
    var
        UseGenFile : record
                        case boolean of
                           true  : (FileOfInterest : ^FOfInt);
                           false : (Offset : integer;
                                    Segment : integer);
                     end;
    begin
    loadAdr(GenFile);
    storexpr(UseGenFile.Offset);
    storexpr(UseGenFile.Segment);
    { Now FileOfInterest can be used }
                    .
                    .
                    .
    end;

    procedure Proc2(var GenFile : file);
    const
        STDW = 183;
    var
        FileOfInterest : ^FOfInt;
    begin
    loadAdr(FileOfInterest);
    loadAdr(GenFile);
    InLineByte(STDW);
    { Now FileOfInterest can be used }
                    .
                    .
                    .
    end;


begin
Proc1(F);
Proc1(F2);
Proc2(F);
Proc2(F2);
end.
```

12. Procedure/Function Parameter Types

12.1 Parameter Lists

PERQ Pascal permits, but does not require, the parameter list of procedures and functions which have been forward declared to be repeated at the site of the actual declaration. If given, the parameter list must match the previous declaration or a compilation error will occur. For redeclaration of function parameters, both the parameter list and the function type must be repeated.

12.2 Function Result Type

PERQ Pascal functions may return any type, with the exception of type FILE.

12.3 Procedures and Functions as Parameters

PERQ Pascal supports passing procedures and functions as parameters to other procedures or functions, as described by the BSI/ISO Pascal Standard [BSI79].

You must put the full description of the procedure parameter in the routine header (just as if it was defined by itself). For example:

```
Procedure EnumerateAll (directory: String; Function for each one
                        (filename: String): boolean; all:boolean);
```

Note that if the procedure is forward declared or in an export section, the routine header for this procedure cannot be repeated

13. Modules

The module facility provides the ability to encapsulate procedures, functions, data and types, as well as supporting separate compilation. Modules may be separately compiled, and intermodule type checking will be performed as part of the compilation process. Unless an identifier is exported from a module, it is local to that module and cannot be used by other modules. Likewise all identifiers referenced in a module must be either local to the module or imported from another module.

Modules do not contain a main statement body. A program is a special instance of a module and conforms to the definition of a program given by the PASCAL User Manual and Report [JW74]. Only a program may contain a main body, and every executable group of modules must contain exactly one instance of a program.

Exporting allows a module to make constants, types, variables, procedures and functions available to other modules. Importing allows a module to make use of the EXPORTS of other modules.

Global constants, types, variables, procedures and functions can be declared by a module to be private (available only to code within the module) or exportable (available within the module as well as from any other module which imports them).

Modules which contain only type and constant declarations cause no run-time overhead; making them ideal for common declarations. It should also be noted that such modules may not be compiled (as errors will be produced), however they may be successfully imported.

131 IMPORTS Declaration

The IMPORTS Declaration specifies the modules which are to be imported into a module. The declaration includes the name of the module to be imported and the file name of the source file for that module. When compiling an import declaration, the source file containing the module to be imported must be available to the compiler.

Note: If the module is composed of several INCLUDE files, only those files from the file containing the program or module heading through the file which contains the word PRIVATE, must be available. (See "Compiler Switches" in this manual.)

The syntax for the IMPORTS declaration is:

<import declaration part> ::= IMPORTS <module name> FROM <file name>;

13.2 EXPORTS Declaration Section

If a program or module is to contain any exports, the EXPORTS Declaration section must immediately follow the program or module heading. The EXPORTS Declaration section is comprised of the word EXPORTS followed by the declarations of those items which are to be exported. These definitions are given as previously specified with one exception: procedure and function bodies are not given in the exports section. Only forward references are given. (See "Declaration Relaxation" in this manual.) (See Chapter 11.2 in the "PASCAL User Manual and Report" [JW74].) The inclusion of "FORWARD;" in the EXPORTS reference is omitted.

The EXPORTS Declaration section is terminated by the occurrence of the word PRIVATE. This signifies the beginning of the declarations which are local to the module. The PRIVATE Declaration section must contain the declarations and bodies for all procedures and functions defined in the EXPORTS Declaration section.

If a program is to contain no EXPORTS Declaration section, the inclusion of PRIVATE following the program heading is optional (PRIVATE is assumed). (Note: A module with no EXPORTS would be useless, since its contents could never be referenced -- it only makes sense for a program not to have any EXPORTS.)

The new syntax for a unit of compilation is:

```
<compilation unit> ::= <module> ¦ <program>

<program> ::= <program heading><module body><statement part>.

<module> ::= <module heading><module body>.

<program heading> ::= PROGRAM <identifier> ( <file identifier>
        {, <file identifier>});

<module heading> ::= MODULE <identifier>;

<module body> ::= EXPORTS <declaration part> PRIVATE
        <declaration part> ¦ PRIVATE <declaration part> ¦
        <declaration part>
```

14.0 Exceptions

PERQ Pascal provides an exception handling facility. Exceptions are typically used for error conditions. There are three steps to using an exception:

1) The exception must be declared.

2) A handler for the exception must be defined.

3) The exception must be raised.

An exception is declared by the word EXCEPTION followed by its name and optional list of parameters. For example:

```
EXCEPTION DivisionByZero(Numerator : integer);
```

Exceptions can be declared anywhere in the declaration portion of a program or module.

The second step is to specify the code to be executed when the exception condition occurs. This is done by defining a handler with the same name and parameter types as the declared exception, for example:

```
HANDLER DivisionByZero(Top : integer);
<block>
```

(See "Declaration Relaxation" in this document for a definition of <block>.) Essentially, a handler looks like a procedure with the word HANDLER substituted for the word PROCEDURE. Handlers may appear in the same places procedures are allowed, with one difference. Handlers cannot be global to a module (they may, however, be global to the main program). The number and type of parameters of a handler must match those of the corresponding exception declaration but the names of the parameters may be different. Multiple handlers can exist for the same exception as long as there is only one per name scope. The exception must be declared before its handler(s).

Raising an exception is analogous to a procedure call. The word RAISE appears before the name and parameters of the exception, for example:

```
RAISE DivisionByZero(N);
```

causes the appropriate handler to execute. The appropriate handler is determined by the current subprogram calling sequence. The run-time stack is searched until a subprogram containing a handler (of the same name) is found. The search starts from the subprogram which issues the RAISE.

For example:

```
program ExampleException;

exception Ex;

handler Ex;
  begin
  writeln('Global Handler for exception Ex');
  end;

procedure Proc1;
  begin
  raise Ex;
  end;

procedure Proc2;
  handler Ex;
    begin
    writeln('Local Handler for exception Ex')
    end;
  begin
  raise Ex;
  Proc1;
  end;

begin
raise Ex;
Proc2;
Proc1;
end.
```

produces the following output:

```
Global Handler for exception Ex
Local Handler for exception Ex
Local Handler for exception Ex
Global Handler for exception Ex
```

Handlers which are already active are not eligible for reactivation. In this case the search continues down the run-time stack until a non-active handler is found. A handler cannot, therefore, invoke itself by raising the same exception it was meant to handle. If a recursive procedure contains a handler, each activation of the procedure has its own eligible handler.

If an exception is raised for which no handler is defined or eligible, the system catches the exception and invokes the debugger. A facility is provided to allow the user to catch such exceptions before the system does. Handlers can be defined for the predefined exception ALL:

```
EXCEPTION ALL(ES,ER,PStart,PEnd : integer);
```

where

1) ES is the system segment number of the exception,

2) ER the routine number of the exception,

3) PStart the stack offset of the first word of the exception's original parameters and,

4) PEnd the stack offset of the word following the original parameters.

Any raised exception that does not have an eligible handler in the same or suceeding level (of the calling sequence), in which an ALL handler is defined, is caught by that ALL handler. The four integer parameter values are calculated by the system and supplied to the ALL handler. Extreme caution should be used when defining an ALL handler, as the handler will also catch system exceptions. All cannot be raised explicitly. The ability to define ALL handlers is intended for "systems hackers" only.

The operating system provides several default handlers. Information on these handlers can be found in the "Program System" and "Module Except" descriptions in the Operating System Manual.

Since exceptions are generally used for serious errors, careful consideration should be given as to whether or not execution should be resumed after an exception is raised. When a handler terminates, execution resumes at the place following the RAISE statement. The handler can, of course, explicitly dictate otherwise. The EXIT and GOTO statements may prove useful here (See "Control Structures" in this manual.)

15.0 Dynamic Space Allocation and Deallocation

The PERQ Pascal Compiler supports the dynamic allocation procedures NEW and DISPOSE defined on page 105 of PASCAL User Manual and Report [JW74], along with several upward compatible extensions which permit full utilization of the PERQ memory architecture.

There are two features of PERQ's memory architecture which require extensions to the standard allocation procedures. First, there are situations which require particular alignment of memory buffers, such as IO operations. Second, PERQ supports multiple data segments from which dynamic allocation may be performed. This facilitates grouping data together which are to be accessed together, which may improve PERQ's performance due to improved swapping. Data segments are multiples of 256 words in size and are always aligned on 256 word boundaries. For further information of the memory architecture and available functions see the documentation on the memory manager.

15.1 NEW

If the standard form of the NEW procedure call is used:

NEW(Ptr{,Tag1,...TagN})

memory for Ptr will be allocated with arbitrary alignment from the default data segment.

The extended form of the NEW procedure call is:

NEW(Segment,Alignment,Ptr{,Tag1,...TagN})

Segment is the segment number from which the allocation is to be performed. This number is returned to the user when creating a new data segment. The value 0 is used to indicate the default data segment.

Alignment specifies the desired alignment; Any power of 2 to 256 (2**0 through 2**8) is permissable. Do not use zero to specify the desired alignment.

If the extended form of NEW is used, both a segment and alignment must be specified; there is no form which permits selective inclusion of either characteristic.

If the desired allocation from any call to NEW cannot be performed, a NIL pointer is usually returned. However, if memory is exhausted, the FULLMEMORY exception may be raised. If the call to NEW fails and raises FULLMEMORY, the user program will abort unless it

includes a handler for FULLMEMORY.

15.2 DISPOSE

DISPOSE is identical to the definition given in PASCAL User Manual and Report [JW74]. Note that the segment and alignment are never given to DISPOSE, only the pointer and tag field values.

16. Single Precision Logical Operations

The PERQ Pascal compiler supports a variety of single precision (INTEGER) logical operations. The operations supported include: and, inclusive or, not, exclusive or, shift and rotate. The syntax for their use resembles that of a function call; however the code is generated inline to the procedure (hence there is no procedure call overhead associated with their use). The syntax for the logical functions are described in the following sections.

16.1 And

```
Function LAND(Val1,Val2: integer): integer;
```

LAND returns the bitwise AND of Val1 and Val2.

16.2 Inclusive Or

```
Function LOR(Val1,Val2: integer): integer;
```

LOR returns the bitwise INCLUSIVE OR of Val1 and Val2.

16.3 Not

```
Function LNOT(Val: integer): integer;
```

LNOT returns the bitwise complement of Val.

16.4 Exclusive Or

```
Function LXOR(Val1,Val2: integer): integer;
```

LXOR returns the bitwise EXCLUSIVE OR of Val1 and Val2.

16.5 Shift

```
Function SHIFT(Value, Distance: integer): integer;
```

SHIFT returns Value shifted Distance bits. If Distance is positive, a left shift occurs, otherwise, a right shift occurs. When performing a left shift, the Least Significant Bit is filled with a 0, and likewise when performing a right shift, the Most Significant Bit is filled with a 0.

16.6 Rotate

```
Function ROTATE(Value, Distance: integer): integer;
```

ROTATE returns Value rotated Distance bits. If Distance is positive a right rotate occurs, otherwise a left rotate occurs. Note that the direction of the ROTATE is the opposite of SHIFT.

17. Input/Output Intrinsics

PERQ's Input/Output intrinsics vary slightly from PASCAL User Manual and Report [JW74].

17.1 REWRITE

The REWRITE procedure has the following form:

REWRITE(F,Name)

F is the file variable to be associated with the file to be written and Name is a string containing the name of the file to be created. EOF(F) becomes true and a new file may be written. The only difference between the PERQ and PASCAL User Manual and Report[JW74] REWRITE is the inclusion of the filename string.

17.2 RESET

The RESET procedure has the following form:

RESET(F,Name)

F is the file variable to be associated with the existing file to be read and Name is a string containing the name of the file to be read. The current file position is set to the beginning of file, i.e. RESET assigns the value of the first element of the file to F^. EOF(F) becomes false if F is not empty; otherwise, EOF(F) becomes true and F^ is undefined.

17.3 READ/READLN

PERQ Pascal supports extended versions of the READ and READLN procedures defined by PASCAL User Manual and Report [JW74]. Along with the ability to read longs, integers (and subranges of integers), reals and characters, PERQ Pascal also supports reading booleans, packed arrays of characters, and strings.

The strings TRUE and FALSE (or any unique abbreviations) are valid input for parameters of type boolean. Mixed upper and lower case are permissible.

If the parameter to be read is a PACKED ARRAY[m..n] of CHAR, then the next n-m+1 characters from the input line will be used to fill the array. If there are fewer than n-m+1 characters on the line, the array will be filled with the available characters, starting at the m'th position, and the remainder of the array will be filled with

blanks.

If the parameter to be read is of type STRING, then the string variable will be filled with as many characters as possible until either the end of the input line is reached or the maximum length of the string is met. If there are not enough characters on the line to fill the entire string, the dynamic length of the string will be set to the number of characters read.

17.4 WRITE/WRITELN

PERQ Pascal provides many extensions to the WRITE and WRITELN procedures defined by PASCAL User Manual and Report [JW74]. Due to the scope of these extensions, the WRITE and WRITELN procedures are completely redefined below:

1. write(p1,...,pn) stands for write(output,p1,...,pn)

2. write(f,p1,...,pn) stands for BEGIN write(f,p1); ... write(f,pn) END

3. writeln(p1,...,pn) stands for writeln(output,p1,...pn)

4. writeln(f,p1,...,pn) stands for BEGIN write(f,p1); ... write(f,pn); writeln(f) END

5. Every parameter pi must be of one of the forms:

   e

   e : e1

   e : e1 : e2

   where e, e1 and e2 are expressions.

6. e is the VALUE to be written and may be of type CHAR, long, integer (or subrange of integer), real, boolean, packed array of char, or string. For parameters of type boolean, one of the strings TRUE, FALSE or UNDEF will be written; UNDEF is written if the internal form of the expression is neither 0 nor 1.

7. e1, the minimum field width, is optional. In general, the value e is written with e1 characters (with preceding blanks). With one exception, if e1 is smaller than the number of characters required to print the given value, more space is allocated; if e is a packed array of char, then only the first e1 characters of the array will be printed.

8. e2, which is optional, is applicable only when e is of type long, integer (or subrange of integer) or real. If e is of type long or integer (or subrange of integer) then e2 indicates the base in which the value of e is to be printed. The valid range for e2 is 2..36 and -36..-2. If e2 is positive, then the value of e is printed as a signed quantity (16-bit twos complement); otherwise, the value of e is printed as a full 16-bit unsigned quantity. If e2 is omitted, the signed value of e is printed in base 10. If e is of type real, then e2 specifies the number of digits to follow the decimal point. The number is then printed in fixed-point notation. If e2 is omitted, then real numbers are printed in floating-point notation.

18. Miscellaneous Intrinsics

18.1 StartIO

STARTIO is a special QCode (See the PERQ QCode Reference Manual) which is used to initiate input/output operations to raw devices. PERQ Pascal supports a procedure, STARTIO, to facilitate generation of the correct QCode sequence for I/O programming. The procedure call has the following form:

```
STARTIO(Unit)
```

where Unit is the hardware unit number of the device to be activated.

18.2 RasterOp

RasterOp is a special QCode which is used to manipulate blocks of memory of arbitrary sizes. It is especially useful for creating and modifying displays on the screen. RasterOp modifies a rectangular area (called the "destination") of arbitrary size (to the bit). The picture drawn into this rectangle is computed as a function of the previous contents of the destination and the contents of another rectangle of the same size called the "source". The functions performed to combine the two pictures are described below.

To allow RasterOp to work on memory other than that used for the screen bitmap, RasterOp has parameters that specify the areas of memory to be used for the source and destination: a pointer to the start of the memory block and the width of the block in words. Within these regions, the positions of the source and destination rectangles are given as offsets from the pointer. Thus position (0,0) would be at the upper left corner of the region, and, for the screen, (767, 1023) would be the lower right. The operating system module Screen exports useful parameters.

The compiler supports a RASTEROP intrinsic which may be used to invoke the RasterOp QCode. The form of this call is:

```
RASTEROP(Function,
         Width,
         Height,
         Destination-X-Position,
         Destination-Y-Position,
         Destination-Area-Line-Length,
         Destination-Memory-Pointer,
         Source-X-Position,
         Source-Y-Position,
         Source-Area-Line-Length,
         Source-Memory-Pointer)
```

Note: the values for the destination precede those for the source.

The arguments to RasterOp are defined below:

"Function" defines how the source and the destination are to be combined to create the final picture stored at the destination. The RasterOp functions are as follows: (Src represents the source and Dst the destination):

Function	Name	Action
0	RRpl	Dst gets Src
1	RNot	Dst gets NOT Src
2	RAnd	Dst gets Dst AND Src
3	RAndNot	Dst gets Dst AND NOT Src
4	ROr	Dst gets Dst OR Src
5	ROrNot	Dst gets Dst OR NOT Src
6	RXor	Dst gets Dst XOR Src
7	RXNor	Dst gets Dst XNOR Src

The symbolic names are exported by the file "Raster.Pas".

"Width" specifies the size in the horizontal ("x") direction of the source and destination rectangles (given in bits).

"Height" specifies the size in the vertical ("y") direction of the source and destination rectangles (given in scan lines).

"Destination-X-Position" is the bit offset of the left side of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Y-Position" is the scan-line offset of the top of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Area-Line-Length" is the number of words which comprise a line in the destination region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Destination-Memory-Pointer" is the 32-bit virtual address of the top left corner of the destination region (it may be a pointer variable of any type). This pointer MUST be quad-word aligned, however. (See "New" in this document for details on buffer alignment.)

"Source-X-Position" is the bit offset of the left side of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Y-Position" is the scan-line offset of the top of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Area-Line-Length" is the number of words which comprise a line in the source region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Source-Memory-Pointer" is the 32-bit virtual address of the top left corner of the source region (it may be a pointer variable of any type). This pointer MUST be quad-word aligned, however. (See "New" in this document for details on buffer alignment.)

18.3 WordSize

The WordSize intrinsic returns the number of words of storage required for any item which has a size associated with it. This includes constants, types, variables and functions. The intrinsic takes a single parameter, the item whose size is desired, and returns an integer.

Note: WordSize generates compile time constants, and hence may be used in constant expressions.

18.4 MakePtr

The MakePtr intrinsic permits the user to create a pointer to a data type given a system segment number and offset. Its use is intended for those who are familiar with the system and are sure of what they are doing. The function takes three parameters. The first two are the system segment number and offset within that segment to be used in creating the pointer, respectively, given as integers. The last parameter is the type of the pointer to be created. MakePtr will return a pointer of the type named by the third parameter.

Note: The next seven intrinsics, MakeVRD, InLineByte, InLineWord, InLineAWord, LoadExpr, LoadAdr and StorExpr, require that the user have knowledge of how the compiler generates code (which will not be discussed here). These intrinsics are intended for "system hacking", and are made available for those who know what they are doing. The programmer who wishes to experiment with these may find the QCode disassembler, QDIS, is very useful to determine if the desired results were produced.

18.5 MakeVRD

MakeVRD is used to load a variable routine descriptor for a procedure or function. (See "Routine Calls and Returns" for a description of LVRD and CALLV in the QCode Reference Manual.) The variable routine descriptor is left on the expression stack of the PERQ, and any further operations must be performed by the user. This procedure takes one parameter, the name of the function or procedure for which the variable routine descriptor is to be loaded. The use of this intrinsic assumes that the programmer is familiar with QCode (primarily a "hacker's" intrinsic).

18.6 InLineByte

InLineByte permits the user to place explicit bytes directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of actual QCodes into a program. InLineByte requires one parameter, the byte to be inserted. The type of this parameter must be either integer or subrange of integer.

18.7 InLineWord

InLineWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of direct QCodes into a program. InLineWord requires one parameter, the word to be inserted. This word

will be inserted immediately as the next two bytes of the code stream (no word alignment is performed). The type of this parameter must be either integer or subrange of integer.

18.8 InLineAWord

InLineAWord permits the user to place explicit words directly into the code stream generated by the compiler. This intrinsic is particulary useful for insertion of direct QCodes into a program. InLineAWord requires one parameter, the word to be inserted. This word is placed on the next word boundary of the code stream. The type of this parameter must be either integer or subrange of integer.

18.9 LoadExpr

The LoadExpr intrinsic takes an arbitrary expression as its parameter and "loads" the value of the expression. The result of the "load" is wherever the particular expression type would normally be loaded (expression stack for scalars, memory stack for sets, etc.).

18.10 LoadAdr

The LoadAdr intrinsic loads the address of an arbitrary data item onto the expression stack. The parameter to LoadAdr, the item whose address is desired, may include array indexing, pointer dereferencing and field selections. The address which is left on the expression stack will be a virtual address if the parameter includes either the use of a VAR parameter or a pointer dereference; otherwise a 20-bit stack offset will be loaded.

18.11 StorExpr

StorExpr stores the single word on top of the expression stack in the variable given as a parameter. The destination for the store operation must not require any address computation; the destination must be a local, intermediate or global variable; it must not be a VAR parameter; if it is a record, a field specification may be given.

18.12 Float

The Float intrinsic converts an integer into a floating point number.

18.13 Stretch

The Stretch intrinsic converts a single precision integer to a double precision integer.

18.15 Shrink

The Shrink intrinsic converts a double precision integer to a single precision integer. If the double precision integer is outside the range of -32768 to +32767, a runtime error occurs.

19. Command Line and Compiler Switches

19.1 Command Line

The PERQ Pascal compiler is invoked by typing a compile command line to the PERQ Operating System. The syntax for the compile command line is:

COMPILE [<InputFile>] [~ <OutputFile>] {</Switch>}.

<InputFile> is the name of the source file to be compiled. The compiler searches for <InputFile>. If it does not find <InputFile>, it appends the extension ".PAS" and searches again. If <InputFile> is still not found, the user will be prompted for an entire command line. If <InputFile> is not specified, the compiler uses for <InputFile> the last file name remembered by the system.

<OutputFile> is the name of the file to contain the output of the compiler. The extension ".SEG" will be appended to <OutputFile> if it is not already present. Note that if <OutputFile> is not specified, the compiler uses the file name from <InputFile>. Then, if the ".PAS" extension is present, it is replaced with the ".SEG" extension, else if the ".PAS" extension is not present, the ".SEG" extension is appended. If <OutputFile> already exists, it will be rewritten.

</Switch> is the name of a compiler switch. All compiler switches specified on the command line must begin with the "/" character. Any number of switches may be specified, and if a switch is specified multiple times, the last occurrence is used. Also, if the /HELP switch is specified, the other information on the command line is ignored. The available switches are defined in the following sections.

19.2 Compiler Switches

PERQ Pascal compiler switches may be set either in a mode similar to the convention described on pages 100-102 of PASCAL User Manual and Report [JW74] or on the command line described above (see above Section, "Command Line"). The first form of compiler switches may be written as comments and are designated as such by a dollar sign character ($) as the first character of the comment followed by the switch (unique abbreviations are acceptable) and possibly switch parameters. The second form is given after the input file specification in the command line preceded by the slash (/) character. The actual switches provided by the PERQ Pascal compiler, although similar in syntax, bear little resemblance to the switches described in PASCAL User Manual and Report [JW74].

The following sections describe the various switches currently supported by the PERQ Pascal Compiler.

19.2.1 File Inclusion

The PERQ Pascal compiler may be directed to include the contents of secondary source files in the compilation. The effect of using the file inclusion mechanism is identical to having the text of the secondary file(s) present in the primary source file (the primary source file is that file which the compiler was told to compile).

To include a secondary file, the following syntax is used:

{$INCLUDE FILENAME}

The characters between the "$INCLUDE" and the "}" are taken as the name of the file to be included (leading spaces and tabs are ignored). The comment must terminate at the end of the filename, hence no other options can follow the filename.

If the file FILENAME does not exist, ".PAS" will be concatenated onto the end of FILENAME, and a second attempt will be made to .find the file.

The file inclusion mechanism may be used anywhere in a program or module, and the results will be as if the entire contents of the include file were contained in the primary source file (the file containing the include directive).

Note: There is no form of this switch for the command line, it may only be used in comment form within a program.

19.2.2 List Switch

The List Switch controls whether or not the compiler generates a program listing of the source text. The default is to not generate a list file. The format for the List switch is:

{$LIST <filename>}

or

/LIST [= <filename>]

where <filename> is the name of the file to be written. The extension ".LST" will be appended to <filename> if it is not already present. If <filename> is not specified, the compiler uses the source file name. If the ".PAS" extension is present, it is replaced with the

".LST" extension, else if the ".PAS" extension is not present, the ".LST" extension is appended. Like the file inclusion mechanism, in the comment form of the switch, the filename is taken as all characters between the "$LIST" and the "}" (ignoring leading spaces and tabs); hence no other options may be included in this comment.

With each source line, the compiler prints the line number, segment number, and procedure number.

19.2.3 Range Checking

This switch is used to enable or disable the generation of additional code to perform checking on array subscripts and assignments to subrange types.

Default value: Range checking enabled

$RANGE+	or /RANGE	enables range checking
$RANGE-	or /NORANGE	disables range checking
$RANGE=		resumes the state of range checking which was in force before the previous $RANGE-or $RANGE+ switch.

If "$RANGE" is not followed by a "+" or "-" , ,then "+" is assumed.

Note that programs compiled with range checking disabled will run slightly faster, but invalid indices will go undetected. Until a program is fully debugged, it is advisable to keep range checking enabled.

19.2.4 Quiet Switch

This switch is used to enable or disable the Compiler from displaying the name of each procedure and function as it is compiled.

Default value: Display of procedure and function names enabled

$QUIET+	or /VERBOSE	enables display of procedure and function names
$QUIET-	or /QUIET	disables display of procedure and function names
$QUIET=		resumes the state of the quiet switch which was in force before the previous $QUIET- or $QUIET+ switch.

if "$QUIET" is not followed by a "+" or "-", then "+" is assumed.

19.2.5 Symbols Switch

This switch is used to set the number of symbol table swap blocks used by the Compiler. As the number of symbol table swap blocks increases, compiler execution time becomes shorter; however physical memory requirements increase (and the Compiler may abort due to insufficient memory). The format for this switch is:

/SYMBOLS = <# of Symbol Table Blocks>

Note: There is no comment form of this switch, it may only be used on a command line.

The default number of symbol table blocks and the maximum number of symbol table blocks are both dependent on the size of memory. For systems with 256k bytes of main memory, the default number of symbol table blocks is 24 and the maximum number of symbol table blocks is 32. Note that you can specify more than 32 symbol table blocks with a 256k byte system, but performance usually degrades considerably. For systems with 512k or 1024k bytes of main memory, the default number of symbol table blocks is 200 and the maximum number of symbol table blocks is also 200.

19.2.6 Automatic RESET/REWRITE

The PERQ Pascal compiler automatically generates a RESET(INPUT) and REWRITE(OUTPUT). This may be disabled if desired with the use of the AUTO switch. The format for this switch is:

Default value: Automatic initialization enabled

$AUTO+ or /AUTO enables automatic initialization

$AUTO- or /NOAUTO disables automatic initialization

If "$AUTO" is not followed by a "+" or "-", then "+" is assumed.

If the comment form of this switch is used, it must precede the BEGIN of the main body of the program.

19.2.7 Procedure/Function Names

The PERQ Pascal compiler generates a table of the procedure and function names at the end of the ".SEG" file, if so directed. This table may be useful for debugging programs. The format for this switch is:

Default value: Name Table is generated

$NAMES+ or /NAMES enables generation of the Name Table

$NAMES- or /NONAMES disables generation of the Name Table

If "$NAMES" is not followed by a "+" or "-", then "-" is assumed.

Note: currently two programs, the debugger and the disassembler, use the information stored in the Name Table.

19.2.8 Version Switch

The Version Switch permits the inclusion of a version string in the first block of the ".SEG" file. This string has a maximum length of 80 characters. Currently this string is not used by any other PERQ software, however, it may be accessed by user programs to identify ".SEG" files. The format for this switch is:

$VERSION <string> or /VERSION = <string> to set the Version string.

When using the $VERSION form of the switch, the version string is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored. If the /VERSION form is used, the version string is terminated by either the end of the command line or the occurrence of a '/' character (hence a '/' may not appear in the version string).

19.2.9 Comment Switch

The Comment Switch permits the inclusion of arbitrary text in the first block of the ".SEG" file. This string has a maximum length of 80 characters. It is particularly useful for including copyright notices in ".SEG" files. The format for this switch is:

$COMMENT <string> or /COMMENT = <string> to set the comment string.

When utilizing the $COMMENT form of the switch, the comment text is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored.

19.2.10 Message Switch

The Message Switch causes the text of the switch to be printed on the user's screen when the switch is parsed by the compiler. It has no effect on the compilation process. The format for this switch is:

$MESSAGE <string> to print <string> on the console during compilation

The message is terminated by the end of the comment or the end of the line. If the comment exceeds a single line, the remainder of the comment is ignored.

Note: There is no command line form for this switch, it may only be used in its comment form.

19.2.11 Conditional Compilation

The PERQ Pascal conditional compilation facility is implemented through the standard switch facility. There are three switches which are used for conditional compilations. The first is the $IFC switch, which has the following form:

{$IFC <boolean expression> THEN}

This switch indicates the beginning of a region of conditional compilation. If the boolean expression, evaluated at compile time, is true, the text to follow is included in the compilation. If the boolean expression evaluates to false, then the text which follows is not included.

The region of conditional compilation is terminated by the $ENDC switch:

{$ENDC}

Upon encountering the $ENDC switch, the state of compilation returns to whatever state was present prior to the most recent $IFC.

The remaining switch is the $ELSEC switch, and it functions much in the same way as the else clause in an IF statement. If the boolean expression of the $IFC switch is true, then the $ELSEC text is ignored, otherwise it is included.

If a $ELSEC switch is used, no $ENDC preceeds the $ELSEC; the $ELSEC signals the end of the $IFC region. A $ENDC is then used to terminate the $ELSEC clause.

Conditional compilations may be nested.

The following are two examples of the conditional compilation mechanism:

```
Const CondSw = TRUE;
PROCEDURE Test;
begin
{$IFC CondSw THEN}
    Writeln('CondSw is true');
{$ENDC}
end { Test };

TYPE Base = record i,j,k:integer end;
{$IFC WORDSIZE(Base) = 3 THEN}
    Cover = array[0..2] of integer
{$ELSEC}
    Cover = array[0..10] of integer
{$ENDC};
```

19.2.12 Errorfile Switch

When the compiler detects an error in a program, it displays error information (file, error number, and the last two lines where the error occurred) on the screen and then requests whether or not to continue. The /ERRORFILE switch overrides this action. When you specify the switch and the compiler detects an error, the error information is written to a file and there is no query of the user. However, the compiler does display the total number of errors encountered on the screen.

The format for this switch is: /ERRORFILE [= <filename>]

where <filename> is the name of the file to be written. The extension ".ERR" will be appended to <filename> if it is not already present. If <filename> is not specified, the compiler uses the source file name. If the ".PAS" extension is present, it is replaced with the ".ERR" extension, else if the ".PAS" extension is not present, the ".ERR" extension is appended.

The error file exists after a compilation if and only if you specify the /ERRORFILE switch and an error is encountered. If the file <filename>.ERR already exists from a previous compilation, it will be rewritten, or deleted in the case of no compiliation errors. This switch allows compilations to be left unattended.

19.2.13 Help Switch

The Help switch provides general information and overrides all other switches. The format is /HELP.

20.0 Quirks and Other Oddities

The following are descriptions of known quirks and problems with the PERQ Pascal compiler. Future releases may correct these problems.

1. FOR loops with an upper bound of greater than 32,766 will never terminate.

2. The last line of any PROGRAM or MODULE must end with a carriage return, or an "Unexpected End of Input" error occurs.

3. Although unique abbreviations are accepted for switches, the following abbreviations cause compilation errors:

Switch	Bad Abbreviation
$ELSEC	$ELSE
$ENDC	$END
$IFC	$IF
$INCLUDE	$IN

4. Procedures and functions which are forward declared (this includes EXPORT declarations) and contain procedure parameters, may not have their parameter lists redeclared at the site of the procedure body.

5. The compiler currently permits the use of an EXIT statement where the routine to be exited from is at the same lexical level as the routine containing the EXIT statement. For example:

```
program Quirk;

    procedure ProcOne;
    begin
    end;

    procedure ProcTwo;
    begin
    exit(ProcOne)
    end;

begin
ProcTwo
end.
```

If there is no invocation of the routine to be

exited on the run-time stack, the PERQ will hang and has to be re-booted.

6. The filename specification given in IMPORTS Declarations must start with an alphabetic character.

7. Record comparisons involving packed records (illegal comparisons) will not be caught unless the word PACKED appears explicitly in the record definition. For example, records with fields of user-defined type Foo, where Foo contains packed information, are considered comparable by the compiler when in actuality they are not.

8. Reals and longs cannot be used together in an expression.

9. The compiler will not detect an error in the definition or use of a set that exceeds set size limitations. If such a set is used, incorrect code will be generated.

10. Many functions that exist for integers (e.g. LAND) are not implemented for longs.

11. The RECAST intrinsic does not work with two-word scalers (for example, LONG) and arrays.

20. References

[BSI79] "BSI/ISO Pascal Standard," Computer, April 1979.

[FP80] "An Implementation Guide to a Proposed Standard for Floating Point", Computer, January 1980

[JW74] K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer Verlag, New York, 1974.

[P*] J. Hennessy and F. Baskett, "Pascal*: A Pascal Based Systems Programming Language," Stanford University Computer Science Department, TRN 174, August 1979.

[UCSD79] K. Bowles, Proceedings of UCSD Workshop on System Programming Extensions to the Pascal Language, Institute for Information Systems, University of California, San Diego, California, 1979.

INDEX

(Entries entirely in upper case are reserved words or predeclared identifiers)

PERQ Operating System Interface

This Manual describes each module in the PERQ Operating System. It gives an abstract for each module, a list of the constants, types, variables, exceptions, procedures, and functions that the module exports, and information about each exported procedure or function.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

TABLE OF CONTENTS

```
module AlignMemory;


AlignMemory - Allocated aligned buffers.
J.  P.  Strait  29 Sep 81.
Copyright (C) Three Rivers Computer Corporation.

Abstract:
     This module is used to allocate buffers which need to be  aligned
     on boundaries that are multiples of 256 words.

Version Number V1.1
exports


  type AlignedBuffer = array[0..0] of array[0..255] of Integer;
       AlignedPointer = ^AlignedBuffer;


procedure NewBuffer( var P: AlignedPointer; S, A: Integer );
exception BadAlignment( A: Integer );
```

```
procedure NewBuffer( var P: AlignedPointer; S, A: Integer );
```

Abstract: This procedure is used to allocate buffers which need to be aligned on boundaries that are multiples of 256 words. A new segment is allocated which is somewhat larger than the desired buffer size. The segment is set to be unmovable so that the alignment can be guaranteed.

Parameters:

- P - Set to point to a new buffer which is aligned as desired.
- S - Desired size of the buffer in 256 word blocks.
- A - Alignment in 256 word blocks. That is, 1 means aligned on a 256 word boundary, 2 means a 512 word boundary, and so on.

Errors: BadAlignment if A is less than one or greater than 256. BadSize (memory manager) if S is less than one or S+A-1 is greater than 256. Other memory manager exceptions raised by CreateSegment.

module AllocDisk;

Written by CMU-people

Abstract:
Allocdisk allocates and deallocates disk pages.

The partition has some number of contiguous pages on it. The number of pages in a partition is specified when the partition is created (using the Partition program). Segments can be created within a partition, e.g. segments may not span partitions.

The entire disk can be thought of as a partition (the Root Partition)

A DiskInformationBlock (DiskInfoBlock or DIB) contains all the fixed information about a disk, including its partion names, locations and sizes. It also contains a table used to locate boot segments

A disk can be 'mounted' which means that its root partition is known to the system as an entry in the DiskTable.

A Partition Information Block (PartInfoBlock or PIB) contains all of the fixed information about a partition,

A partition can also be 'mounted', and this is usually done as part of mounting the disk itself. Partitions mounted are entries in the PartTable

Within a partition, segments are allocated as doubly linked lists of pages

The Free List of a segment is a doubly linked list of free pages

This module maintains this list, as well as the DeviceTable and PartTable It contains procedures for mounting and dismounting disks and partitions, as well as allocating and deallocating space within a partition.

When allocating pages, the module updates the PartInfoBlock every MaxAllocs calls on AllocDisk

Since the system may crash some time between updates, the pointers and free list size may not be accurate.

When a partition is mounted, the pointers are checked to see if they point to free pages. If not, the head of the pointer is found by looking at the "filler" word of the block the free head does point to (which presumably was allocated after the last update of PartInfoBlock). The filler word has a short pointer to the next "free" page, and forms a linked list to the real free list header. Likewise, if the Free tail does not have a next

pointer of 0, a deallocate is presumed to have been done since a PartInfoBlock update, and NextAdr pointers are chased to find the real end of the Free List.

```
Version Number V2.8
{******************} exports {****************************}

imports Arith from Arith;
imports ReadDisk from ReadDisk;

const
  MAXDISKS      =  2;  {Floppy and HardDisk}
  MAXPARTITIONS = 10;  {Maximum number of mounted partitions}
  MAXPARTCHARS  =  8;  {how many characters in a partition name}

type

  PartString   = string[MAXPARTCHARS];

  DeviceRecord = record                  {entry in the DeviceTable}
                   InfoBlk: DiskAddr;    {where the DiskInfoBlock is}
                   InUse  : boolean;     {this DeviceTable entry is valid}
                   RootPartition: PartString {name of this disk}
                 end;

  PartRecord   = record                          {entry in the PartTable}
                   PartHeadFree : DiskAddr; {pointer to Head of Free List}
                   PartTailFree : DiskAddr; {pointer to tail of Free List}
                   PartInfoBlk  : DiskAddr; {pointer to PartInfoBlock}
                   PartRootDir  : DiskAddr; {pointer to Root Directory}
                   PartNumOps   : integer;  {how many operations done since
                                             last update of PartInfoBlock}
                   PartNumFree  : FSBit32;  {HINT of how many free pages}
                   PartInUse    : boolean;  {this entry in PartTable is valid}
                   PartMounted  : boolean;  {this partition is mounted}
                   PartDevice   : integer;  {which disk this partition is in}
                   PartStart    : DiskAddr; {Disk Address of 1st page}
                   PartEnd      : DiskAddr; {Disk Address of last page}
                   PartKind     : PartitionType; {Root or Leaf}
                   PartName     : PartString {name of this partition}
                 end;


var
  DiskTable   : array [0..MAXDISKS-1]  of DeviceRecord;
  PartTable   : array [1..MAXPARTITIONS] of PartRecord;


procedure InitAlloc;  {initialize the AllocDisk module, called during boot}

procedure DeviceMount(disk: integer);  {mount a disk}

procedure DeviceDismount(disk : integer);  {dismount a disk}

function  MountPartition(name : string) : integer;  {mount a partion,
                                                     return PartTable index}
```

```
·edure DismountPartition(name : string);  {dismount a partition}

inction  FindPartition(name : string) : integer;  {given a partion name, look
                                                    for it in PartTable, return index}

inction  AllocDisk(partition : integer) : DiskAddr;  {allocate a free page
                                                       from a partition}

·ocedure DeallocDisk(addr : DiskAddr);  {return a page to the free list}

·ocedure DeallocChain(firstaddr,lastaddr : DiskAddr; numblks : integer);
                                {return a bunch of pages to free list}

inction  WhichPartition(addr : DiskAddr) : integer;  {given a Disk Address,
                                                      figure out which partition it is in}

·ocedure DisplayPartitions;  {print the PartTable}

:ception NoFreePartitions;

    Abstract:  Raised  when  too many partitions are accessed at one
         time.  The
              limit is MAXPARTITIONS.

:ception BadPart(msg, partName: String);

    Abstract:  Raised when there is something wrong with a partition.
         This
              means that the Scavenger should be run.

    Parameters: msg is the problem  and  partName  is  the  partition
         name.  Print
              error message as: WriteLn('** ',msg,' for ',partName);

:ception PartFull(partName: String);

    Abstract:  Raised when there are no free blocks in a partition to
         be
              allocated.  This  means  that  some  files  should  be
              deleted and then the Scavenger should be run.

    Parameters: partName is the full partition
```

Procedure InitAlloc; Abstract Initialize the AllocDisk module

Side Effects Sets Initialized to a magic number; sets all InUse and PartInUse to false

Procedure DeviceMount(disk : integer);

Abstract: Mount the device specified by disk if not already mounted

Parameters: Disk is a device; it should be zero for HardDisk and 1 for Floppy

Environment: Expects DiskTable to be initialized

Side Effects: Sets the DiskTable for device; loads PartTable with Part names on dev

Errors: Error if no free partition slots in PartTable;

NOTE: No mention is made if device has partitions with names same as those already loaded

Procedure DisplayPartitions;

Abstract: Displays information about the current partitions on the screen

Environment: Assumes PartTable and DiskTable set up;

Calls: AddrToField; IntDouble, WriteLn;

Procedure DeviceDismount(disk : integer);

Abstract: Removes device disk (0 or 1) from DiskTable and removes all its partitions

Parameters: Disk is a device (0= HardDisk; 1=Floppy)

Side Effects: Sets DiskTable[disk].InUse to false and removes all of disk's partitions

Calls: DismountPartition

Function FindPartition(name : string) : integer;

Abstract: Searches through partition table looking for a partition named name; if found; returns its index in the table;

Parameters: name is partition name of form "dev:part>" or ":part>" or "part>" where the final ">" is optional in all forms. If dev isn't specified then searches through all partition names. If dev is specified; then only checks those partitions on that device; name may be in any case

Returns: index in PartTable of FIRST partition with name name (there may be more than one partition with the same name in which case it uses the oldest one) or zero if not found or name malformed;

Calls: UpperEqual

Design: No device name specified is signaled by disk=MAXDISKS; otherwise disk is set to be the device which the device part of name specifies

Function MountPartition(name : string) : integer;

Abstract: Searches for partition name in part table and mounts it if not mounted already; tries to read the head and tail of free list to see if valid

Parameters: name is partition name of form "dev:part>" where "dev" and ">" are optional

Returns: index in part Table of partition for name or zero if not found

Side Effects: if not mounted already, then reads in PartInfoBlk and sets partTable fields; tries to read the head and tail of free list to see if valid

Errors: if no free slots for partition then Raises NoFreePartitions if can't find free list head or tail then Raises BadPart

Calls: FindPartition

Procedure DismountPartition(name : string);

Abstract: Removes partition name from PartTable

Parameters: name is partition name of form "dev:part>" where "dev" and ">" are optional

Side Effects: Writes out part information in table if partition InUse and mounted

Calls: UpdatePartInfo, ForgetAll

Function AllocDisk(partition: integer) : DiskAddr;

Abstract: Allocate a free block from partition

Parameters: Partition is the partition index to allocate the block from

Returns: Disk Address of newly freed block;

Side Effects: Updates the partition info to note block freed; changes header in buffer of block; writes new head of free list with its next and prev fields set to zero and its filler set to next free block; decrements PartNumFree

Errors: Raises PartFull if no free blocks in partition Raises BadPart if free list inconsistent

Calls: ReadHeader, ChangeHeader, FlushDisk, UpdatePartInfo

Function WhichPartition(addr : DiskAddr) : integer;

Abstract: Given a disk address; find the partition it is in

Parameters: addr is a disk address

Returns: index of partition addr falls inside of or zero if none

Calls: DoubleBetween

NOTE: DOESN'T CHECK IF ENTRY IN TABLE IS MOUNTED OR INUSE (***Bug***???)

Procedure DeallocDisk(addr : DiskAddr);

Abstract: Returns block addr to whatever partition it belongs to

Parameters: addr is block to deallocate

Side Effects: adds addr to free list; increments PartNumFree

Calls: AddToTail, WhichPartition, UpdatePartInfo

Procedure DeallocChain(firstaddr,lastaddr : DiskAddr; numblks : integer);

Abstract: Deallocates a chain of blocks

Parameters: firstAddr and lastAddr are addresses of blocks to deallocate (inclusive) and numBlks is number of blocks to free

Side Effects: Frees first and last addr using AddToTail; middle blocks not changed

Calls: AddToTail, ChangeHeader, WhichPartition, DoubleAdd, FlushDisk, UpdatePartInfo, DoubleInt

NOTE: No checking is done to see if numBlks is correct

```
module Arith;


Needed until Pascal compiler supports type long.
Copyright (C) 1980 Carnegie-Mellon University
Version Number V2.2
exports

imports FileDefs from FileDefs;  { to get FSBitnn }

type
  MyDouble = packed record
             case integer of
               1:
                 (
                   Lsw : integer;
                   Msw : integer
                 );
               2:
                 (
                   Ptr : FSBit32
                 );
               3:
                 (
                   Byte0 : FSBit8;
                   Byte1 : FSBit8;
                   Byte2 : FSBit8;
                   Byte3 : FSBit8
                 )
             end;

function DoubleAdd(a,b : FSBit32) : FSBit32;
function DoubleSub(a,b : FSBit32) : FSBit32;
function DoubleNeg(a   : FSBit32) : FSBit32;
function DoubleMul(a,b : FSBit32) : FSBit32;
function DoubleDiv(a,b : FSBit32) : FSBit32;
function DoubleInt(a   : integer) : FSBit32;
function IntDouble(a   : FSBit32) : integer;
function DoubleBetween(a,start,stop : FSBit32) : boolean;

function DoubleMod(a,b : FSBit32) : FSBit32;
function DoubleAbs(a   : FSBit32) : FSBit32;

function DblEql(a,b : FSBit32) : boolean;
function DblNeq(a,b : FSBit32) : boolean;
function DblLeq(a,b : FSBit32) : boolean;
function DblLes(a,b : FSBit32) : boolean;
function DblGeq(a,b : FSBit32) : boolean;
function DblGtr(a,b : FSBit32) : boolean;
```

```
Function DoubleAdd( a,b : FSBit32) : FSBit32;
     Abstract: Adds two doubles together
     Parameters: a and b are doubles to add
     Returns: a+b

Function DoubleSub(a,b : FSBit32) : FSBit32;
     Abstract: Subtracts b from a
     Parameters: a and b are doubles
     Returns: a-b
     Design: a+(-b)

Function DoubleNeg(a : FSBit32 ) : FSBit32;
     Abstract: Does a two-s complement negation of argument
     Parameters: a is number to negate
     Returns: -a

Function DoubleAbs(a : FSBit32 ) : FSBit32;
     Abstract: Does an absolute value of argument
     Parameters: a is number to abs
     Returns: |a|

Function DoubleMul(a,b : FSBit32) : FSBit32;
     Abstract: Multiplies a and b
     Parameters: a and b are doubles
     Returns: a*b
```

```
Function DoubleDiv(a,b : FSBit32) : FSBit32;
```

Abstract: Divides a by b

Parameters: a and b are doubles

Returns: a/b

```
Function DoubleMod(a,b : FSBit32) : FSBit32;
```

Abstract: Mods a by b

Parameters: a and b are doubles

Returns: a mod b

```
Function DoubleInt(a : integer) : FSBit32;
```

Abstract: converts a into a double

Parameters: a is integer

Returns: double of a; if a is negative then does a sign extend

```
Function IntDouble(a : FSBit32 ) : integer;
```

Abstract: returns the low word of a

Parameters: a is a double

Returns: low word

Errors: Micro-code raises OvflLI (in Except) if a will not fit in one word

```
Function DoubleBetween(a,start,stop : FSBit32) : boolean;
```

Abstract: determines whether a is between start and stop (inclusive)

Parameters: a is a double to test; start is low double and stop is high

Returns: true if a >= start and a <= stop else false

```
function DblEql(a,b : FSBit32): boolean;
     Abstract: determines whether a = b
     Parameters: a and b are doubles
     Returns: true if a = b; else false

function DblNeq(a,b : FSBit32): boolean;
     Abstract: determines whether a <> b
     Parameters: a and b are doubles
     Returns: true if a <> b; else false

function DblLeq(a,b : FSBit32) : boolean;
     Abstract: determines whether a <= b
     Parameters: a and b are doubles
     Returns: true if a <= b; else false

function DblLes(a,b : FSBit32) : boolean;
     Abstract: determines whether a < b
     Parameters: a and b are doubles
     Returns: true if a < b; else false

function DblGeq(a,b : FSBit32) : boolean;
     Abstract: determines whether a >= b
     Parameters: a and b are doubles
     Returns: true if a >= b; else false

function DblGtr(a,b : FSBit32) : boolean;
     Abstract: determines whether a > b
     Parameters: a and b are doubles
     Returns: true if a > b; else false
```

```
module Clock;
```

Clock - Perq clock routines.
J. P. Strait 1 Feb 81.

Abstract:

Clock implements the Perq human-time clock. Times are represented internally by a TimeStamp record which has numeric fields for Year, Month, Day, Hour, Minute, and Second. Times may also be expressed by a string of the form YY MMM DD HH:MM:SS where MMM is a three (or more) letter month name and HH:MM:SS is time of day on a 24 hour clock.

The clock module exports routines for setting and reading the current time as either a TimeStamp or a character string, and exports routines for converting between TimeStamps and strings.

```
Version Number V1.6
exports


imports GetTimeStamp from GetTimeStamp;


const ClockVersion = '1.6';


type TimeString = String;


procedure SetTStamp( Stamp: TimeStamp );
procedure SetTString( String: TimeString);
procedure GetTString( var String: TimeString );
procedure StampToString( Stamp: TimeStamp; var String: TimeString );
procedure StringToStamp( String: TimeString; var Stamp: TimeStamp);

Exception BadTime;
```

Abstract: Raised when a string passed does not represent a valid time

```
procedure SetTStamp( Stamp: TimeStamp );
```

Abstract: Sets time to be time specified by Stamp

Parameters: stamp is new time

SideEffects: Changes current time

```
procedure SetTString( String: TimeString);
```

Abstract: Sets time to be time specified by String

Parameters: string is the string of the new time

SideEffects: Changes current time

Errors: Raises BadTime is string is invalid (malformed or illegal time)

```
procedure GetTString( var String: TimeString );
```

Abstract: Returns the current time as a string

Parameters: string is the string to be set with the current time

```
procedure StampToString( Stamp: TimeStamp; var String: TimeString );
```

Abstract: Returns a string for the time specified by stamp

Parameters: stamp is time to get string for; string is set with time represented by stamp

```
procedure StringToStamp( String: TimeString; var Stamp: TimeStamp );
```

Abstract: Converts string into a time stamp

Parameters: string is the string containing time; stamp is stamp set with time according to string

Errors: Raises BadTime is string is invalid (malformed or illegal time)

```
Module CmdParse;


     This module provides a number of routines to  help  with  command
     parsing.

     Written by Don Scelza    April 30, 1980

     Copyright (C) 1980 - Three Rivers Computer Corporation
Version Number V3.6
{********************} Exports {********************}

Const CmdPVersion = '3.5';
      MaxCmds = 30;
      MaxCString = 255;
      CCR = Chr(13);  {same as standard CR}
      CmdChar = Chr(24);
      CmdFileChar = Chr(26);

Type CString = String[MaxCString];
     CmdArray = Array[1..MaxCmds] Of String;

Procedure CnvUpper(Var Str:CString);  {*** USE ConvUpper IN PERQ_String***}
Function UniqueCmdIndex(Cmd:CString; Var CmdTable: CmdArray;
                        NumCmds:Integer): Integer;
Procedure RemDelimiters(Var Src:CString; Delimiters:CString;
                        Var BrkChar:CString);
Procedure GetSymbol(Var Src,Symbol:CString; Delimiters:CString;
                    Var BrkChar:CString);

Function NextID(var id: CString; var isSwitch: Boolean): Char;
Function NextIDString(var s, id: CString;var isSwitch: Boolean): Char;

Type pArgRec = ^ArgRec;
     ArgRec = RECORD
                name: CString;
                next: pArgRec;
              END;

     pSwitchRec = ^SwitchRec;
     SwitchRec = RECORD
                   switch: CString;
                   arg: CString;
                   correspondingArg: pArgRec;
                   next: pSwitchRec;
                 END;

Function ParseCmdArgs(var inputs, outputs: pArgRec; var switches: pSwitchRec;
                      var err: String): boolean;
Function ParseStringArgs(s: CString; var inputs, outputs: pArgRec;
                      var switches: pSwitchRec; var err: String): boolean;
Procedure DstryArgRec(var a: pArgRec);
Procedure DstrySwitchRec(var a: pSwitchRec);

Type pCmdList = ^CmdListRec;
     CmdListRec = RECORD
```

```
                cmdFile: Text;
                isCharDevice: Boolean;
                next: pCmdList;
                seg: Integer;
              END;

ocedure InitCmdFile(var inF: pCmdList; seg: Integer);
nction DoCmdFile(line: CString; var inF: pCmdList; var err: String): boolean;
ocedure ExitCmdFile(var inF: pCmdList);
ocedure ExitAllCmdFiles(var inF: pCmdList);
ocedure DstryCmdFiles(var inF: pCmdList);

nction RemoveQuotes(var s: CString): boolean;

pe ErrorType = (ErBadSwitch, ErBadCmd, ErNoSwParam, ErNoCmdParam, ErSwParam,
                ErCmdParam, ErSwNotUnique, ErCmdNotUnique, ErNoOutFile,
                ErOneInput, ErOneOutput, ErFileNotFound, ErDirNotFound,
                ErIllCharAfter, ErCannotCreate, ErAnyError, ErBadQuote);

ocedure StdError(err: ErrorType; param: CString; leaveProg: Boolean);
nction NextString(var s, id: CString; var isSwitch: Boolean): Char;
```

```
Procedure InitCmdFile(var inF: pCmdList; seg: Integer);
```

Abstract: Initializes inF to be a valid Text File corresponding to the keyboard. This must be called before any other command file routines. The application should then read from inF^.cmdFile. E.g. ReadLn(inFile^.cmdFile, s); or while not eof(inFile^.cmdFile) do ... Use popup only if inF^.next = NIL (means no cmd File). Is a fileSystem file if not inF^.isCharDevice. InF will never be NIL. The user should not modify the pCmdList pointers; use the procedures provided.

Parameters: InF - is set to the new command list.

seg - the segment number to allocate the command file list out of. If the application doesn't care, use 0. This is useful for programs like the Shell that require the list of command files to exist even after the program terminates. For other applications, use 0.

```
Function DoCmdFile(line: CString; var inF: pCmdList; var err: String):
    boolean;
```

Abstract: This procedure is meant to handle an input line that specifies that a command file should be invoked. The application will find a line that begins with an @ and will call this procedure passing that line. This procedure maintains a stack of command files so that command files can contain other command files. Be sure to call InitCmdFile before calling this procedure.

Parameters: line - the command line found by the application. It is OK if it starts with an @ but it is also OK if it doesn't.

inF - the list of command files. This was originally created by InitCmdFile and maintained by these procedures. If the name is a valid file, a new entry is put on the front of inF describing it. If there is an error, then inF is not changed. In any case, inF will always be valid.

err - if there is an error, then set to a string describing the error, complete with preceding '** '. If no error, then set to ''. The application can simply do: if not DoCmdFile(s, inF, err) then WriteLn(err);

Returns: True if OK, or false if error.

```
Procedure ExitCmdFile(var inF: pCmdList);
```

Abstract: Remove top command file from list. Call this whenever come to end of a command file.

Parameters: InF - the list of command files. It must never be NIL. The top entry is removed from inF unless attempting to remove last command file, when it is simply re-initialized to be the console. It is OK to call this routine even when at the last entry of the list. Suggested use: While EOF(inF^.cmdFile) do ExitCmdFile(inF);

```
Procedure ExitAllCmdFiles(var inF: pCmdList);
```

Abstract: Remove all command file from list. Use when get an error or a ^SHIFT-C to reset all command files.

Parameters: InF - the list of command files. It must never be NIL. All entries but the last are removed.

```
Procedure DstryCmdFiles(var inF: pCmdList);
```

Abstract: Removes all command files from list.

Parameters: InF - the list of command files. All entries are removed and InF set to NIL.

```
Function NextID(var id: CString; var isSwitch: Boolean): Char;
```

Abstract: Gets the next word off UsrCmdLine and returns it. It is OK to call this routine when UsrCmdLine is empty (id will be empty and return will be CCR) This procedure also removes comments from s (from to end of line is ignored). This is exactly like NextIDString except it uses the UsrCmdLine by default.

WARNING: It is a bad idea to mix calls to NextID and RemDelimiters/GetSymbol since the latter 2 may change UsrCmdLine in a way that causes NextID to incorrectly report that an id is not a switch. This procedure also appends a CCR to the end of UsrCmdLine so it should not be printed after this procedure is called.

Parameters:

id - set to the next word on UsrCmdLine. If there are none, then id will be the empty string.

isSwitch - tells whether the word STARTED with a slash "/". The slash is not returned as part of the name.

Results: The character returned is the next "significant" character after the id. The possible choices are "=" "," " " "~" CCR. CCR is used to mean the end of the line was hit before a significant character. If there are spaces after the id and then one of the other break characters defined above, then the break character is returned. If there is a simply another id and no break characters, then SPACE is returned.

SideEffects: Puts a CCR at the end of UsrCmdLine so it is a bad idea to print UsrCmdLine after NextID is called the first time. Removes id and separators from front of UsrCmdLine. The final character is also removed.

Function NextIDString(var s, id: CString; var isSwitch: Boolean): Char;

Abstract: Gets the next word off s and returns it. It is OK to call this routine when s is empty (id will be empty and return will be CCR) This procedure also removes comments from s (from to end of line is ignored). This is exactly like NextID except it allows the user to specify the string to parse.

WARNING: It is a bad idea to mix calls to NextIDString and RemDelimiters/GetSymbol since the latter 2 may change s in a way that causes NextIDString to incorrectly report that an id is not a switch.

Parameters:

s - String to parse. Changed to remove id and separators from front. The final character is also removed.

id - set to the next word in string. If there are none, then id will be the empty string.

isSwitch - tells whether the word STARTED with a slash "/". The slash is not returned as part of the name.

Results: The character returned is the next "significant" character after the id. The possible choices are "=" "," " " "~" CCR. CCR is used to mean the end of the line was hit before a significant character. If there are spaces after the id and then one of the other break characters defined above, then the break character is returned. If there is a simply another id and no break characters, then SPACE is returned.

Function NextString(var s, id: CString; var isSwitch: Boolean): char;

Abstract: Gets the next word off s and returns it. It is OK to call this routine when s is empty (id will be empty and return will be CCR) This procedure also removes comments from s (from to end of line is ignored). The character after id is NOT removed from s. This is like NextIDString except the character is removed in NextIDString.

Parameters:

s - String to parse. Changed to remove id and separators from front. Final character is NOT removed.

id - set to the next word in string. If there are none, then id will be the empty string.

isSwitch - tells whether the word STARTED with a slash "/". The slash is not returned as part of the name.

Results: The character returned is the next "significant" character after the id. This character remains at the front of s. The possible choices are "=" "," " " "~" CCR. CCR is used to mean the end of the line was hit before a significant character. If there are spaces after the id and then one of the other break characters defined above, then the break character is returned. If there is a simply another id and no break characters, then SPACE is returned.

Function RemoveQuotes(var s: CString): boolean;

Abstract: Changes all quoted quotes ('') into single quotes (').

Parameters: s - string to remove quotes from. It is changed.

Returns: true if all ok. False if a single quote ended the string. In this case s still contains that quote.

Function ParseCmdArgs(var inputs, outputs: pArgRec; var switches: pSwitchRec; var err: String): boolean;

Abstract: Parses the command line assuming standard form. The command should be removed from the front using NextId before ParseCmdArgs is called.

Parameters:

inputs - set to list describing the inputs. There will always be at least one input, although the name may be empty.

outputs - set to list describing the outputs. There will always be at least one output, although the name may be empty.

switches - set to the list of switches, if any. Each switch points to the input or output it is attached to. This may be NIL if the switch appears before any inputs. If a global switch, the application can ignore this pointer. If a switch is supposed to be local, the application can search for each input and output through the switches looking for the switches that correspond to this arg. Switches may be NIL if there are none.

err - set to a string describing the error if there is one. This string can simply be printed. If no error, then set to ''.

Returns: false if there was a reported error so the cmdLine should be rejected. In this case, the pArgRecs should be Destroyed anyway.

Function ParseStringArgs(s: CString; var inputs, outputs: pArgRec; var switches: pSwitchRec; var err: String): boolean;

Abstract: Parses the string assuming standard form. The command should be removed from the front using NextId before ParseCmdArgs is called.

Parameters:
s - the string to parse.

inputs - set to list describing the inputs. There will always be at least one input, although the name may be empty.

outputs - set to list describing the outputs. There will always be at least one output, although the name may be empty.

switches - set to the list of switches, if any. Each switch points to the input or output it is attached to. This may be NIL if the switch appears before any inputs. If a global switch, the application can ignore this pointer. If a switch is supposed to be local, the application can search for each input and output through the switches looking for the switches that correspond to this arg. Switches may be NIL if there are none.

err - set to a string describing the error if there is one. This string can simply be printed. If no error, then set to ''.

Returns: false if there was a reported error so the string should be rejected. In this case, the pArgRecs should be Destroyed anyway.

Procedure DstryArgRec(var a: pArgRec);

Abstract: Deallocates the storage used by a ArgRec list.

Parameters: a - the head of the list of ArgRecs to deallocate. It is set to NIL. OK if NIL before call.

Procedure DstrySwitchRec(var a: pSwitchRec);

Abstract: Deallocates the storage used by a SwitchRec list.

Parameters: a - the head of a list of pSwitchRecs to deallocate. It is set to NIL. OK if NIL before call.

```
Procedure   StdError(err:   ErrorType;   param:   CString;   leaveProg:
     Boolean);
```

Abstract: Prints out an error message with a parameter and then optionally exits the user program.

Parameters: err - the error type found
leaveProg - if true then after reporting error, raises ExitProg to return to the shell. If false then simply returns
param - parameter for the error. The message printed is:

ErBadSwitch	- "** <PARAM> is an invalid switch."
ErBadCmd	- "** <PARAM> is an invalid command."
ErNoSwParam	- "** Switch <PARAM> does not take any arguments."
ErNoCmdParam	- "** Command <PARAM> does not take any arguments."
ErSwParam	- "** Illegal parameter for switch <PARAM>."
ErCmdParam	- "** Illegal parameter for command <PARAM>."
ErSwNotUnique	- "** Switch <PARAM> is not unique."
ErCmdNotUnique	- "** Command <PARAM> is not unique."
ErNoOutFile	- "** <PARAM> does not have any outputs."
ErOneInput	- "** Only one input allowed for <PARAM>."
ErOneOutput	- "** Only one output allowed for <PARAM>."
ErFileNotFound	- "** File <PARAM> not found."
ErDirNotFound	- "** Directory <PARAM> does not exist."
ErIllCharAfter	- "** Illegal character after <PARAM>."
ErCannotCreate	- "** Cannot create file <PARAM>."
ErBadQuote	- "** Cannot end a line with Quote."
ErAnyError	- "<PARAM>"

```
Procedure CnvUpper(Var Str:CString);
```

Abstract: This procedure is used to convert a string to uppercase.

Parameters: Str is the string that is to be converted.

Side Effects: This procedure will change Str. ****WARNING**** THIS PROCEDURE WILL SOON BE REMOVED. USED THE PROCEDURE ConvUpper IN PERQ_STRING

```
Function UniqueCmdIndex(Cmd:CString; Var CmdTable: CmdArray;
    NumCmds:Integer): Integer;
```

Abstract: This procedure is used to do a unique lookup in a command table.

Parameters: Cmd - the command that we are looking for.

CmdTable - a table of the valid commands. The first valid command in this table must start at index 1.

NumCmds - the number of valid command in the table.

Results: This procedure will return the index of Cmd in CmdTable. If Cmd was not found then return NumCmds + 1. If Cmd was not unique then return NumCmds+2.

```
Procedure RemDelimiters(Var Src:CString; Delimiters:CString; Var
    BrkChar:CString);
```

Abstract: This procedure is used to remove delimiters from the front of a string.

Parameters:

Src - the string from which we are to remove the delimiters.
Delimiters - a string that contains the characters that are to be considered delimiters.
BrkChar - will hold the character that we broke on.

Side Effects: This procedure will change both Src and BrkChar.

```
Procedure GetSymbol(Var  Src,Symbol:CString;  Delimiters:CString;  Var
     BrkChar:CString);
```

Abstract: This procedure is used to remove the first symbol from the beginning of a string.

Parameters:

Src - the string from which we are to remove the symbol.

Symbol - a string that is used to return the next symbol.

Delimiters - a string that defines what characters are to be considered delimiters. Any character in this string will be used to terminate the next symbol.

BrkChar - used to return the character that stopped the scan.

Side Effects: This procedure will remove the first symbol from Src and place it into Symbol. It will place the character that terminated the scan into BrkChar.

```
odule Code;


ode.Pas - Common definitions for the Linker and Loader.
.  P.  Strait          10 Feb 81.  Rewritten as a module.
opyright (C) Three Rivers Computer Corporation, 1981.

ostract:

    Code.Pas defines constants and types shared by the Linker and the
    Loader.  These include definitions of the run file and of offsets
    in the stack segment.

    Design:  When  the  format  of run files is changed, the constant
    RFileFormat must also be changed.  This is necessary  to  prevent
    the procedures which read run files from failing.

xports

 imports GetTimeStamp from GetTimeStamp;


onst    CodeVersion = '1.7';
        RFileFormat = 2;

        QCodeVersion = 3;         { Current QCode Version Number }
        FileLength   = 100;       { max chars in a file name }
        SegLength    = 8;         { max chars in a segment name }
        StackLeader  = 2;         { number of leader words in stack before }
                                  { XSTs (must be even)                    }
                                  { Currently contains initial TP and GP   }
        DefStackSize = #20;       { default stack segment size (in blocks) }
        DefHeapSize  = #4;        { default heap segment size (in blocks) }
        DefIncStack  = #4;        { default stack size increment (in blocks) }
        DefIncHeap   = #4;        { default heap size increment (in blocks) }
        FudgeStack   = #2000;     { fudge space between system and user GDB's }
                                  { this must hold all loader variables at }
                                  { maximum configuration in LoadStack }

        CommentLen   = 80;        { the length of comment and version str in seg}


ype
        SNArray = packed array[1..SegLength] of Char;  { segment name }

        pFNString = ^FNString;

        FNString = String[FileLength];  { file name }

        QVerRange = 0..255;             { range of QCode version numbers }


        SegHint = record case Integer of
```

```
            1: (Fid    : Integer;       { file id }
                Update: TimeStamp);     { update time }
            2: (Word1 : Integer;
                Word2 : Integer;
                Word3 : Integer)
            end;


    pSegNode = ^SegNode;

    pImpNode = ^ImpNode;


    { Segment information record:}

    SegNode = record
              SegId     : SNArray;      { segment name }
              RootNam   : pFNString;    { file name without .Pas or .Seg }
              Hint      : SegHint;      { hint to the segment file }
              GDBSize   : integer;      { size of this segment's GDB }
              XSTSize   : integer;      { size of this segment's XST }
              GDBOff    : integer;      { StackBase offset to GDB }
              ISN       : integer;      { segment number inside Linker }
              CodeSize  : integer;      { number of blocks in .Seg file }
              SSN       : integer;
              UsageCnt  : integer;
              ImpList   : pImpNode;
              Next      : pSegNode
              end;


    { Import information record }

    ImpNode = record
              SId       : SNArray;      { name of imported segment }
              FilN      : pFNString;    { file name of imported segment }
              XGP       : integer;      { global pointer of import }
              XSN       : integer;      { internal number of import }
              Seg       : pSegNode;
              Next      : pImpNode
              end;


    { Run file: }

    RunElement = (RunHeader,SysSegment,UserSegment,Import,SegFileNames);

    RunInfo = record   { run header }
              RFileFormat:integer;
              Version:    integer;
              System:     boolean;
              InitialGP:  integer;
              CurOffset:  integer;
              StackSize:  integer;
              StackIncr:  integer;
              HeapSize:   integer;
```

```
            HeapIncr:   integer;
            ProgramSN:  integer;
            SegCount:   integer
            end;

    RunFileType = file of Integer;


    { Segment file: }

    pSegBlock = ^SegBlock;

    SegBlock = packed record case boolean of        { .SEG file definition }
            { first block: }
            true: (ProgramSegment: boolean;
                   SegBlkFiller  : 0..127;
                   QVersion      : QVerRange;
                   ModuleName    : SNArray;
                   FileName      : FNString;
                   NumSeg        : integer;
                   ImportBlock   : integer;
                   GDBSize       : integer;
                   Version       : String[CommentLen];
                   Comment       : String[CommentLen]);
            false:(Block: array[0..255] of integer)
            end;

    CImpInfo = record case boolean of   { Import List Info - as generated }
                                        { by the compiler }
                true: ( ModuleName: SNArray;  { module identifier }
                        FileName:   FNString  { file name }
                      );
                false:( Ary: array [0..0] of integer)
             end;

    SegFileType = file of SegBlock;
```

```
module ControlStore;


ControlStore - Load and call routines in the PERQ control-store.
J. P. Strait  ca. July 80.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     The ControlStore module exports types defining the format of PERQ
     micro-instructions and procedures to load and call routines in
     tne control-store.

Version Number V1.2
exports


type MicroInstruction = { The format of a micro-instruction as produced by
                          the micro-assembler. }
     packed record case integer of
       0: (Word1: integer;
           Word2: integer;
           Word3: integer);
       1: (Jmp: 0..15;
           Cnd: 0..15;
           Z:   0..255;
           SF:  0..15;
           F:   0..3;
           ALU: 0..15;
           H:   0..1;
           W:   0..1;
           B:   0..1;
           A:   0..7;
           Y:   0..255;
           X:   0..255);
       2: (JmpCnd: 0..255;
           Fill1:  0..255;
           SFF:    0..63;
           ALU0:   0..1;
           ALU1:   0..1;
           ALU23:  0..3)
       end;

     MicroBinary = { The format of a micro-instruction and its address as
                     produced by the micro-assembler. }
       record
         Adrs: integer;
         MI:   MicroInstruction
         end;

     TransMicro = { The format of a micro-instruction as needed by the WCS
                    QCode. }
       packed record case integer of
         0: (Word1: integer;
             Word2: integer;
             Word3: integer);
         1: (ALU23: 0..3;
```

```
            ALU0:   0..1;
            W:      0..1;
            ALU1:   0..1;
            A:      0..7;
            Z:      0..255;
            SFF:    0..63;
            H:      0..1;
            B:      0..1;
            JmpCnd:0..255)
        end;

    MicroFile = file of MicroBinary;   { A file of micro-instructions.  }


procedure LoadControlStore( var F: MicroFile );
procedure LoadMicroInstruction( Adrs: integer; MI: MicroInstruction );
procedure JumpControlStore( Adrs: integer );
```

```
procedure LoadControlStore( var F: MicroFile );
```

Abstract: Loads the contents of a MicroFile into the PERQ control-store. The file whould be opened (with Reset) before calling LoadControlStore. It is read to EOF but not closed; thus it should be closed after calling LoadControlStore.

Parameters:
F - The MicroFile that contains the micro-instructions to be loaded.

```
procedure LoadMicroInstruction( Adrs: integer; MI: MicroInstruction );
```

Abstract: Loads a single micro-instruction into the PERQ control-store.

Parameters:
Adrs - The control store address to be loaded.
MI - The micro-instruction to be loaded.

```
procedure JumpControlStore( Adrs: integer );
```

Abstract: Transfers control of the PERQ micro engine to a particular address in the control-store.

Note 1: Values may not be loaded onto the expression stack before calling JumpControlStore. If you wish to pass values through the expression stack, the following code whould be used rather than calling LoadControlStore.

```
LoadExpr(   LOr(   Shift(Adrs,8),   Shift(Adrs,-8)   )   );
InLineByte( #277 );          the JCS QCode *)
```

Note 2: Microcode called by JumpControlStore should terminate with a "NextInst(0)" microcode jump instruction.

Parameters:
Adrs - The address to jump to.

```
)S D.6 Interface                                    module DiskIO
; Feb 82

)dule DiskIO;


)stract:

    This module implements the basic low level operations to disk
    devices.  It services the Hard Disk and the Floppy.  When dealing
    with the floppy here, the structures on the hard disk are mapped
    to the structures on the floppy.
)rsion Number V3.13
)**********} exports {**************************}

)ports Arith from Arith;
)ports FileDefs from FileDefs;
)ports IOErrors from IOErrors;

)nst
   HARDNUMBER          = 0;              {device code of Shugart Disk}
   FLOPPYNUMBER        = 1;              {device code of FloppyDisk}

   {a Disk Address can be distinguished from a Segment Address by the
     upper two bits (in 32 bits).  These bits have a nonzero code to
     which disk the address is part of}

   RECORDIOBITS        = #140000;        {VirtualAddress upper 16 bits of disk}
   DISKBITS            = RECORDIOBITS + (HARDNUMBER*(#20000));
   FLOPBITS            = RECORDIOBITS + (FLOPPYNUMBER*(#20000));

   {The following definitions tell how many entries there are in the
    three pieces of the random index.  The first piece (Direct)
    are blocks whose DiskAddresses are actually contained in the
    Random Index (which is part of the FileInformationBlock).
    The second section has a list of blocks each of which contain 128 Disk
    Addresses of blocks in the file, forming a one level indirect addressing
    scheme.
    For very large files, the third section (DblInd) has DiskAddresses of
    blocks which point to other blocks which contain 128 DiskAddresses
    of blocks in the file, forming a two level indirect scheme.}

   DIRECTSIZE          =  64; { Entries in FIB of blocks directly accessable }
   INDSIZE             =  32; { Entries in FIB of 1 level indirect blocks }
   DBLINDSIZE          =   2; { Entries in FIB of 2 level indirect blocks }

   FILESPERDIRBLK      = 16;  { 256 / SizeOf(DirEntry) }
   NUMTRIES            = 15;  { number of tries at transfer before aborting }

)pe
   {Temporary segments go away when processes are destroyed,
    Permanent segments persist until explicitly destroyed
    Bad Segments are not well formed segments which are not
    readable by the Segment system}

   SpiceSegKind  = (Temporary, Permanent, Bad);
   PartitionType = (Root, UnUsed, Leaf); {A Root Partition is a device}
   DeviceType    = (Winch12, Winch24, FloppySingle, FloppyDouble,
                    UnUsed1, UnUsed2);
```

```
    MyDble = Array [0..1] of integer;

    DiskCheatType = record
                      case integer of
                        1: (
                             Addr     : DiskAddr
                           );
                        2: (
                             Dbl      : MyDble { should be IO.Double but don't
                                                 import IO in export section }
                           );
                        3: (
                             Seg      : SegID
                           );
                        4: (
                             Lng      : FSBit32
                           )
                       end;

    { A directory is an ordinary file which contains SegIDs of files
      along with their names.  Directories are hash coded by file name
      to make lookup fast.  They are often sparse files (ie contain
      unallocated blocks between allocated blocks).  The file name is a
      SimpleName, since a directory can only contain entries for
      files within the partition (and thus device) where the directory
      itself is located }

    DirEntry   = packed record
                   InUse    : boolean; {true if this DirEntry is valid}
                   Deleted  : boolean; {true if entry deleted but not expunged}
                   Archived : boolean; {true if entry is on backup tape}
                   UnUsed   : 0..#17777;   {reserved for later use}
                   ID       : SegID;
                   Filename : SimpleName
                 end;


    DiskBuffer = packed record
                   case integer of
                     1: (
                          Addr : array [0..(DISKBUFSIZE div 2)-1] of DiskAddr
                        );
                     2: (
                          IntData : array [0..DISKBUFSIZE-1] of FSBit16
                        );
                     3: (
                          ByteData : packed array [0..DISKBUFSIZE*2-1] of FSBit
                        );
  {4 is format of the FileInformationBlock; the FIB has Logical Block -1 }
                     4: (

                          FSData     : FSDataEntry;

                          {The Random Index is a hint of the DiskAddresses
                           of the blocks that form the file.
```

```
                    It has three parts as noted above.  Notice
                    that all three parts are always there, so
                    that even in a very large file, the first
                    DIRECTSIZE blocks can be located quickly
                    The blocks in the Random index have logical
                    block numbers that are negative.  The logical
                    block number of Indirect[0] is -2 (the FIB is -1)
                    the last possible block's number is
                    -(INDSIZE+DBLINBDSIZE+1)}

                  Direct      : array [0..DIRECTSIZE-1] of DiskAddr;
                  Indirect    : array [0..INDSIZE-1]    of DiskAddr;
                  DblInd      : array [0..DBLINDSIZE-1] of DiskAddr;

                  SegKind     : SpiceSegKind;

                  NumBlksInUse : integer; {segments can have gaps,
                                           block n may exist when block
                                           n-1 has never been allocated.
                                           NumBlksInUse says how many
                                           data blocks are actually used
                                           by the segment}
                  LastBlk     : FSBit16;  {Logical Block Number of
                                           largest block allocated}
                  LastAddr    : DiskAddr; {DiskAddr of LastBlk }
                  LastNegBlk  : FSBit16;  {Logical Block Number of
                                           largest pointer block
                                           allocated}
                  LastNegAddr: DiskAddr   {Block number of LastNegBlk}
                );
:5 is the format of the DiskInformationBlock}
              5: (

                  {The Free List is a chain of free blocks linked
                  by their headers }

                  FreeHead    : DiskAddr; {Hint of Block Number of the
                                           head of the free list}
                  FreeTail    : DiskAddr; {Hint of Block Number of the
                                           tail of the free list}
                  NumFree     : FSBit32;  {Hint of how many blocks
                                           are on the free list}
                  RootDirID   : SegID;    {where to find the Root
                                           Directory}
                  BadSegID    : SegID;    {where the bad segment is}

                  {when booting, the boot character is indexed into
                   the following tables to find where code to be
                   boot loaded is found }

                  BootTable   : array [0..25] of DiskAddr; {qcode}
                  InterpTable: array [0..25] of DiskAddr; {microcode}
                  PartName    : packed array [1..8] of char;
                  PartStart   : DiskAddr;
                  PartEnd     : DiskAddr;
                  SubParts    : array [0..63] of DiskAddr;
```

```
                        PartRoot    : DiskAddr;
                        PartKind    : PartitionType;
                        PartDevice  : DeviceType
                      );
 {6 is the format of a block of a Directory}
                    6: (
                        Entry       : array [0..FILESPERDIRBLK-1] of DirEntry
                      )
                end;

    ptrDiskBuffer = ^DiskBuffer;

    Header      = packed record             {format of a block header}
                    SerialNum : DiskAddr; {Actually has the SegID of the file}
                    LogBlock  : integer;  {logical block number}
                    Filler    : integer;  {holds a hint to a candidate
                                           for the FreeHead}
                    PrevAdr   : DiskAddr; {Disk Address of the next block in
                                           this segment}
                    NextAdr   : DiskAddr; {Disk Address of the previous block ir
                                           this segment}
                  end;

    ptrHeader   = ^Header;

    DiskCommand= (DskRead, DskWrite, DskFirstWrite, DskReset, DskHdrRead,
                  DskHdrWrite);  {last ones for error reporting}

var
    DiskSegment : integer;                  {a memory segment for DiskIO}

procedure InitDiskIO;  {initialize DiskIO, called at boot time}

procedure ZeroBuffer(ptr : ptrDiskBuffer);  {write zeroes in all words of the
                                             buffer.  When reading an unallocate
                                             block, Zeros are returned in the
                                             buffer}

function WhichDisk(addr : DiskAddr) : integer;  {Tells you which disk number a
                                                 DiskAddr is on}

function AddrToField(addr : DiskAddr) : integer;  {gives you a one word short
                                                   address by taking the lower
                                                   byte of the upper word and
                                                   the upper byte of the lower
                                                   word.  The upper byte of the
                                                   upper word can't have any
                                                   significant bits for the 12
                                                   or 24 megabyte disks.  The
                                                   lower byte of the lower word
                                                   is always zero (since a disk
                                                   address is a page address
                                                   which is 256 words
                                                  }
```

```
inction FieldToAddr(disk: integer; fld : integer) : DiskAddr;
                                                    { Makes a DiskAddr out of a
                                                      short address and a disk
                                                      number
                                                    }


·ocedure DiskIO(addr : DiskAddr; ptr : ptrDiskBuffer;
                hptr : ptrHeader; dskcommand : DiskCommand); {Do a disk
                                                              operation, if
                                                              errors occur,
                                                              exits via
                                                              DiskError}


inction LogAddrToPhysAddr(addr : DiskAddr) : DiskAddr;
                                    {translate a Logical Disk Address (used
                                     throughout the system) to and from a
                                     physical Disk Address (the kind the disk
                                     contoller sees) Logical Disk Addresses
                                     use a sequential numbering system
                                     Physical Disk Addresses have a
                                     Cylinder-Head-Sector system This routine
                                     calls MapAddr (a private routine which
                                     does the translation) Map Addr
                                     implements interlace algorithm}

  tion PhysAddrToLogAddr(disk : integer; addr : DiskAddr) : DiskAddr;

inction LastDiskAddr(DevType : DeviceType) : DiskAddr; {Gets the Disk Address
                                                        of the last possible
                                                        page on the device}


inction NumberPages(DevType : DeviceType) : FSBit32;  {Return the number of
                                                       pages on a device}


·ocedure DiskReset;  {Reset the disk controller and recalibrate the actuater}

inction TryDiskIO(addr : DiskAddr;  ptr : ptrDiskBuffer;
                  hptr : ptrHeader;  dskcommand : DiskCommand;
                  numTries: integer) : boolean;
                                    {Try a disk operation, but, return false
                                     if error occurred
                                    }

:eption DiskFailure(msg: String; operation: DiskCommand; addr: DiskAddr;
                    softStat: integer);
:eption DiskError(msg: String);
:eption BadDevice;


  ErrorCnt : Array[IOEFirstError..IOELastError] of integer;
```

```
Procedure InitDiskIO;

     Abstract: Initializes the DiskIO package

     SideEffects: Creates a new segment (DiskSegment). Sets
          Initialzed.

     Calls: CreateSegment

Function NumberPages(DevType:DeviceType) : FSBit32;

     Abstract: Returns number of pages on a disk

     Parameters: DevType: a device code

Function LastDiskAddr(DevType:DeviceType) : DiskAddr;

     Abstract: Returns Disk Address of last possible page on a disk

     Parameters: DevType: a device code

     Calls: NumberOfPages

Procedure ZeroBuffer(ptr : ptrDiskBuffer);

     Abstract: clear buffer to zeroes

     Parameters:
          ptr - a pointer to a disk buffer

Function WhichDisk(addr : DiskAddr) : integer;

     Abstract: Return disk device of a DiskAddr

          Parameters addr a DiskAddr that you want the device code for
```

```
Function AddrToField(addr : DiskAddr) : integer;
```

Abstract: return a short (one word) disk address by taking the low byte of the most significant word of the Disk Address and the upper byte of the least significant word

Parameters:
addr - the Disk Address of to get a short address for

```
Function FieldToAddr(disk: integer; fld : integer) : DiskAddr;
```

Abstract: Return a DiskAddress given a short address and the disk device code

Parameters:
disk - the disk device code
fld - the short address (returned by AddrToField at some point)

```
Function LogAddrToPhysAddr(addr : DiskAddr) : DiskAddr;
```

Abstract: Convert a Logical DiskAddress to a Physical Disk Address

Parameters:
addr - the disk address to be converted

Calls: MapAddr

```
Function PhysAddrToLogAddr(disk : integer; addr : DiskAddr) :
DiskAddr;
```

Abstract: Translate a Physical Disk Address to a logical disk address

Parameters:
disk - Device code of disk
addr - Physical address to translate

Calls: UnMapAddr

```
Procedure DiskReset;
```

Abstract: Reset the Hard Disk and recalibrate the drive

Calls: UnitIO

Design: Assuming that the disk is at track 201, seek in one track until the Track 0 bit in the disk status is set Then force the microcode to believe that this is track 0, and reset the disk controller again

```
Procedure DiskIO(addr: DiskAddr; ptr: ptrDiskBuffer; hptr: ptrHeader;
    dskcommand: DiskCommand);
```

Abstract: Calls DoDiskIO with DieOnError set true; does 15 retries (3 recalibrates)

Parameters: see DoDiskIO

```
function TryDiskIO(addr : DiskAddr; ptr : ptrDiskBuffer; hptr :
    ptrHeader; dskcommand : DiskCommand; numTries: integer) :
    boolean;
```

Abstract: Calls DoDiskIO with DieOnError set false

Parameters: see DoDiskIO

Returns: true if tranfer went ok, false if errors occurred

)dule Dynamic;

/namic - Perq dynamic memory allocation and de-allocation.
. P. Strait 1 Jan 80.
)pyright (C) Three Rivers Computer Corporation, 1980, 1981,1982.

)stract:
Dynamic implements Pascal dynamic allocation - New and Dispose. Memory of a given size with a given alignment may be allocated from any data segment. If the data segment is full (doesn't contain enough free memory to allocate), the segment is increased in multiples of the segment's increment size until there is enough free memory to allocate. Similarly, memory that was once allocated may be deallocated.

Design: Free memory within each segment is linked into a circular freelist in order of address. Each free node is at least two words long and is of the form

record Next: Integer; Length: Integer; Rest: 2*Length - 2 words end;

Where Next*2 is the address of the next free node and Length*2 is the number of free words.

sion Number V1.4
)rts

)nst DynamicVersion = '1.4';

)ports Memory from Memory;

'ocedure NewP(S: SegmentNumber; A: integer; var P: MMPointer; L: integer);
'ocedure DisposeP(var P: MMPointer; L: integer);

```
procedure DisposeP( var P: MMPointer; L: integer );
```

Abstract: Deallocate memory.

Parameters:
- P - Pointer to the memory.
- L - Length in words, 0 represents a length of 2**16. If L is odd, L+1 words are de-allocated.

Errors: NilPointer if P is nil. BadPointer 1) if the Offset part is odd. 2) if Offset+Length > size of segment. 3) if the node to be Dispose overlaps some node that is already free. 4) if the segment is not InUse or not a DataSegment.

```
procedure NewP( S: SegmentNumber; A: integer; var P: MMPointer; L:
    integer );
```

Abstract: Allocate memory.

Parameters:
- S - Number of segment to allocate from. 0 means the default data segment.
- A - Alignment of node in words relative to beginning of segment, 0 represents an alignment of 2**16. if A is odd, A+1 is used as the alignment.
- P - Set to point to the memory that was allocated. If the data segment is full and cannot be increased, P is set to nil.
- L - Length in words, 0 represents a length of 2**16. If L is odd, L+1 words are allocated.

Errors: FullSegment if the segment has reached its maximum size and there isn't enough room for the node. FullMemory if NewP tries to expand the segment, but there enough physical memory to do so. UnusedSegment if S is not InUse. NotDataSegment if S is not a DataSegment.

)dule Ether10IO;

)stract:
This module provides the client interface to the 10 Mbaud Ethernet microcode.

Written by: Don Scelza

{**********} Exports {**********}

)is module provides the raw I/O interface to the Three Rivers Computer ;hernet system. The procedures in this module allow the client to send and :ceive packets on the net.

)r details of the Physical and Data Link layers of the network see the)cument:

)e Ethernet A Local Area Network Data Link Layer and Physical Layer)ecifications

:C - Intel - XEROX

details on the Three Rivers hardware interface to the network see:

;hernet Interface Programmers Guide

·adeep Reddy

)r details on the interface presented, to this module, by the Ethernet icrocode see the file:

;her10.Micro

)nald A. Scelza

)llowing is some general information about the client interface presented y the Ethernet microcode and this module:

; is possible to always have a receive pending. If a send command is :ecuted while a receive is pending the internal state of the interface is ;ved in a register save area in memory. This is done by saving the VA of)e DCB for the receive. After the send has completed the receive state is :loaded and the receive is restarted.

) addition to the ability to do a Receive followed by a Send, it is also)ssible to do multiple Receives. The Receives are linked using the NextDCB ·ld of the Ethernet DCB. When a Receive completes the next Receive in the n is started.

)mmand information for the Ethernet driver is provided in an Ethernet :vice Control Block, DCB. All data areas referenced by pointers in the DCB

as well as the DCB itself must be LOCKED in memory until the request has completed. They can NOT be moved. The best way to do this is to mark the segment that the buffers are allocated from as UnMovable. This will allow the memory manager to place the buffers in a convient place in memory before they are locked down.

The Ethernet driver needs to have a four (4) word area of memory in which it can save registers. A pointer to this area of memory is provided by the BuffPtr when a Reset command is executed. Once the Reset command has been processed this register save area can NOT be moved. To change the register save area another Reset command must be executed.

The Ethernet DCB must be unmovable while the command is pending.

To wait for the completion of a command it is possible to spin on the Command-In-Progress bit in the status block. This bit will be cleared when the requested command has been completed.

After a receive the Bits field of the status block has the number of bits that were received. To translate this into the number of data bytes you must perform a number of operations. First divide it by 8. This will give the number of bytes that were received. If the number is not evenly divisible by 8 then there was a transmissintializeon error. After the division you must subtract the number of bytes in the header and the CRC. There is a total of 20 bytes in these two portions of the packet.

```
nports SystemDefs from SystemDefs;
  >rts System from System;


Define the types and variables used by the Network stuff.
}

/pe

   {
   { These are the valid commands for the Ethernet interface.
   { }

   EtherCommand = (EReset, EReceive, EPromiscuousReceive, ESend);

   {
   { An Ethernet address is 48 bits long.  It is made up of 6
   { octets or in our case 3 words.
   { }

   EtherAddress = packed record   { An address on the net is 48 bits }
       High: integer;
       Mid:  integer;
       Low:  integer;
       end;

   {
   { This record defines an Ethernet status block.  The first
   { 15 bits of the block are defined by the hardware interface.
   { The 16th bit of the first word and the second word are defined by the
   { Ethernet microcode.
   {
   { Alignment:    Double word.
   { Locked:       Yes.
   { }

   EtherStatus = packed record      { The status record.  }
       CRCError:   boolean;         { There was a CRC error.}
       Collision:  boolean;         { There was a collision }
       RecvTrans:  boolean;         { 0 - receive has finished.  1 for trans.  }
       Busy:       boolean;         { The interface is bust.  }
       UnUsed4:    boolean;
       ClockOver:  boolean;         { The microsecond clock overflowed.  }
       PIP:        boolean;         { There is a Packet In Progress.  }
       Carrier:    boolean;         { There is traffic on the net.  }
       RetryTime:  0..15;
       UnUsed12:   boolean;
       UnUsed13:   boolean;
       SendError:  boolean;         { Could not send packet after 16 tries.  }
       CmdInProg:  boolean;         { There is a command pending.  }
       BitsRecv:   integer;         { Number of bits that were received.  }
       end;

   {
   { This record defines the header for an Ethernet transfer.
```

```
    {
    { Alignment:    8 word.
    { Locked:       Yes.
    { }

    EtherHeader = packed record
        Dest:       EtherAddress;
        Src:        EtherAddress;
        EType:      Integer;        { Type field defined by XEROX }
        end;

    {
    { This record provides the definition of an EtherBuffer.
    {
    { Alignment:    1k word.
    { Locked:       Yes.
    { }

    EtherBuffer = array [0..749] of integer;

    {
    { Define all of the pointers that we need.
    { }

    pEtherStatus = ^EtherStatus;
    pEtherBuffer = ^EtherBuffer;
    pEtherHeader = ^EtherHeader;
    pEtherDCB = ^EtherDCB;

    {
    { This is the definition of an Ethernet Device Control Block.
    {
    { Alignment:    Quad word.
    { Locked:       Yes.
    { }


    EtherDCB = packed record
        HeadPtr:    pEtherHeader;
        BuffPtr:    pEtherBuffer;
        StatPtr:    pEtherStatus;
        Cmd:        EtherCommand;
        BitCnt:     Integer;            { Total bits in buffer and header
        NextDCB:    pEtherDCB;
        end;

    {
    { This is the definition that is used to create the register save
    { area.
    {
    { Alignment:    Double word.
    { Locked:       Yes.
    { }

    EtherRegSave = record
        RecvDCB:    pEtherDCB;
```

```
        SendDcb:    pEtherDCB;
        end;

    pEtherRegSave = ^EtherRegSave;



    {
    { Following are the definitions that are used to deal with the
    { micro-second clock.
    { }

    {
    { The microsecond clock takes a two word combined control and
    { status block.  The first word of the block gives the number
    { of microseconds to be loaded into the clock.
    { The second word provides the status information from the
    { clock.  Once a clock command has been started it is
    { possible to spin on the CmdInProgress bit in the control
    { block.  When the bit is cleared the specified number of
    { micro-seconds has elapsed.
    {
    { Alignment:    Double word.
    { Locked:       Yes.
    { }

type

    uSClkDCB = packed record
        uSeconds: integer;
        UnUsed0:   boolean;
        UnUsed1:   boolean;
        UnUsed2:   boolean;
        UnUsed3:   boolean;
        UnUsed4:   boolean;
        UnUsed5:   boolean;
        UnUsed6:   boolean;
        UnUsed7:   boolean;
        UnUsed8:   boolean;
        UnUsed9:   boolean;
        UnUsed10:  boolean;
        UnUsed11:  boolean;
        UnUsed12:  boolean;
        UnUsed13:  boolean;
        UnUsed14:  boolean;
        CmdInProg: boolean;
        end;

    puSClkDCB = ^uSClkDCB;
```

```
{
{ Define the constants for the address block supplied to Three Rivers by
{ Xerox.
{
{ High 16 bits (2 octets) are 02 1C (Hex).
{ Next 8 bits (1 octet) is 7C (Hex).
{
{ The low order byte of the second PERQ word as well as the third PERQ word
{ are Three Rivers defined.  Currently the low order byte of the second word
{ is used to define the type of interface.  The valid values are:
{
{       0 - Interface is on an IO option board.
{       1 - Interface is on the IO board
{ }


const

    TRCCAdrMid = 31744;          { 7C hex in the high order 8 bits.  }
    TRCCAdrHigh = 540;           { 02 1C hex.  }
    EBoardOption = 0;            { The interface is on an I/O Option board.  }
    EBoardIO = 1;                { The interface is on the I/O board.  }


{
{ These are some other useful constants.
{ }

const

    MinDataBytes = 46;           { Smallest number of data bytes in a packet.  }
    MaxDataBytes = 1500;         { Largest number of data bytes in a packet.  }
    NumDCBs = 16;                { The number of DCBs, commands, possible at }
                                 { a single time }


{
{ These are the procedures exported by this module.
{ }

procedure E10Init;
procedure E10IO(Cmd: EtherCommand;  Header: pEtherHeader;  Buff: pEtherBuff;
                Stat: pEtherStatus;  Bytes: Integer);
procedure E10Wait(Stat: pEtherStatus);
procedure E10Reset(MyAdr: EtherAddress);
function  E10DataBytes(RecvBits: Integer): Integer;
function  E10GetAdr: EtherAddress;
procedure E10State(var NumSend, NumReceive: Integer);
procedure E10WIO(Cmd: EtherCommand;  Header: pEtherHeader;
                Buff: pEtherBuff;  Stat: pEtherStatus;  Bytes: Integer);
```

```
{
{ These are the exceptions that may be raised by this module.
{ }

exception E10NInited;
```

Abstract:
This exception will be raised if any procedures in this package are called before E10Init.

```
exception E10NReset;
```

Abstract:
This exception will be raised if any transfer commands are executed before a E10Reset is done.

```
exception E10ByteCount;
```

Abstract:
This exception is raised if a byte count passed to this interface is not in the valid range. The number of data bytes in an Ethernet packet must be in the range 46 <-> 1500, (MinDataBytes <-> MaxDataBytes).

```
exception E10DByteError;
```

Abstract:
This exception is raised if the number of Bits passed to E10DataBytes does not form a valid packet.

```
exception E10BadCommand;
```

Abstract:
This exception is raised if a bad command is given to any of the routines in this package.

```
exception E10TooMany;
```

Abstract:
This exception is raised if more than NumDCBs commands are executed at any time.

```
exception E10STooMany;
```

Abstract:
This exception is raised if more the client tries to execute more than one send.

```
exception E10ReceiveDone(Stat: pEtherStatus);
```

Abstract:
This exception is raised when a receive command has finished. It is raised by the Pascal level interrupt routine for the net.

Parameters:
Stat will be set to the status pointer of the command that finished.

procedure E10Init;

Abstract: This procedure is used to initialize the Ethernet module. It must be called before any other procedure in this package are used.

Side Effects: This procedure will allocate any memory used by this module.

procedure E10IO(Cmd: EtherCommand; Header: pEtherHeader; Buff: pEtherBuff; Stat: pEtherStatus; Bytes: Integer);

Abstract: This procedure is used to start an Ethernet I/O operation and return.

Parameters: Cmd is the command that is to be executed.

Header is a pointer to an Ethernet header block. The client must fill in all fields of this header.

Buff is a pointer to the buffer that is to be sent or filled.

Stat is a pointer to a status block for use during this command.

Bytes is the number of data bytes that are to be transferred. This value must be between 46 and 1500

Exceptions: E10NInited: Raised if this procedure is called before EtherInit.

E10NReset: Raised if this procedure is called before EReset.

E10ByteCount: Raised if Bytes is not in the valid range.

E10BadCommand: This is raised if the command passed is not Send, Receive or PromisciousReceive.

E10TooMany: is raised if too many commands are executed at a given time.

E10STooMany: is raised if more than one send command is executed.

```
procedure E10Wait(Stat: pEtherStatus);
```

Abstract: This procedure will wait for the completion of some Ethernet request.

Parameters: Stat is the pointer to the EtherStatus that was provided when the command was initiated.

Exceptions: E10NInited: Raised if this procedure is called before E10Init.

E10NReset: Raised if this procedure is called before EReset.

```
procedure E10Reset(MyAdr: EtherAddress);
```

Abstract: This procedure is used to reset the Ethernet interface.

Parameters: MyAdr is an Ethernet address block that contains the address of this this interface. Once this address is set any packets that have a Source field that matches MyAdr will be received by the network interface. If there are no Receives pending then the packet will be dumped.

Exceptions: E10NInited: Raised if this procedure is called before EtherInit.

```
function E10DataBytes(RecvBits: Integer): Integer;
```

Abstract: This procedure is used to obtain the number of data bytes that are in a packet that was received over the network.

Parameters: RecvBits is the number of bits that were in the packet. This value will come from the BitsRecv field of the status block.

Results: This function will return the number of data bytes that were in the packet.

Exceptions: E10NInited: Raised if this procedure is called before EtherInit.

E10NReset: Raised if this procedure is called before EReset.

E10DByteError: Raised if the numebr of bits in the packet was not a multiple of 8 or if the number of data bytes was less than MinDataBytes.

function E10GetAdr: EtherAddress;

Abstract: This function will return the address of this machine.

Exceptions: E10NInited: Raised if this procedure is called before E10Init.

E10NReset: Raised if this procedure is called before EReset.

procedure E10State(var NumSend, NumReceive: integer);

Abstract: This procedure is used to return the internal state of the Ethernet interface.

Parameters: NumSend will be set to the number of Sends that are pending.

NumReceive will be set to the number of receives that are pending.

Exceptions: E10NInited: Raised if this procedure is called before E10Init.

E10NReset: Raised if this procedure is called before EReset.

procedure E10WIO(Cmd: EtherCommand; Header: pEtherHeader; Buff: pEtherBuff; Stat: pEtherStatus; Bytes: Integer);

Abstract: This procedure is used to start an Ethernet I/O operation and wait for it to complete.

Parameters: Cmd is the command that is to be executed.

Header is a pointer to an Ethernet header block. The client must fill in all fields of this header.

Buff is a pointer to the buffer that is to be sent or filled.

Stat is a pointer to a status block for use during this command.

Bytes is the number of data bytes that are to be transfered. This value must be between 46 and 1500

Exceptions: E10NInited: Raised if this procedure is called before E10Init.

E10NReset: Raised if this procedure is called before EReset.

E10ByteCount: Raised if Bytes is not in the valid range.

E10BadCommand: This is raised if the command passed is not

Send, Receive or PromiscuousReceive.

E10TooMany: is raised if too many commands are executed at a given time.

E10STooMany: is raised if more than one send command is executed.

```
module EtherInterrupt;


Abstract:
     This module provides the interrupt service for the 10 MBaud
     ethernet.

     Written by: Don Scelza.

     Copyright (C) Three Rivers Computer Corporation, 1981

{**********************} Exports {**********************}


imports Ether10IO from Ether10IO;

var
    StackPointer: Integer;
    DCBStack: array[1..NumDCBs] of pEtherDCB;
    RListHead, RListTail, SListHead: pEtherDCB;
    SendsPosted, RecvsPosted: Integer;

function PopDCB: pEtherDCB;
procedure PushDCB(Ptr: pEtherDCB);
procedure E10Srv;
```

```
function PopDCB: pEtherDCB;
```

Abstract: Get the next free DCB from the stack.

Results: Return a pointer to the next free DCB.

Side Effects: Move the stack pointer.

```
procedure PushDCB(Ptr: pEtherDCB);
```

Abstract: Push a free DCB onto the DCB stack.

Parameters: Ptr is a pointer to the DCB that is to be pushed onto the stack.

Side Effects: Move the stack pointer.

```
procedure E10Srv;
```

Abstract: This is the 10 megabaud Ethernet interrupt routine.

Exceptions: This procedure will raise E10ReceiveDone if a receive has completed.

```
module Except;


Except - Perq Pascal Exception Routines.
J. P. Strait  10 Dec 80.
Copyright (C) Three Rivers Computer Corporation, 1980.

Abstract:
     Module Except provides the following things:

     1) Definitions of the microcode generated exceptions.

     2) A procedure to tell the microcode which segment number these
     exceptions are defined in.

     3) The default handler of all exceptions. The compiler enables
     this handler in every main program.

     4) A Pascal routine to search the stack in when an exception is
     raised.

     Design: The file Except.Dfs is included into Perq.Micro as well
     as into this module. It defines routine numbers for the
     exceptions generated by the microcode. Note that there must be
     agreement between these constants and the routine numbers of the
     exception definitions. No program checks these--if you add or
     remove exception definitions you must be sure to update
     Except.Dfs in the appropriate way.

     The routine number of RaiseP is also defined in Except.Dfs as 0.
     Since the microcode must know this, it is strongly suggested that
     it not be modified.

     The routine number of InitExceptions is not needed by the
     compiler or Perq.Micro, but it has been assigned routine number 1
     so that its number will not change when new exceptions are
     defined. This means that new exceptions may be defined without
     requiring that the operating system be re-linked.

Version Number V2.9
exports


const ExceptVersion = '2.9';


procedure RaiseP( ES, ER, PStart, PEnd: Integer );
procedure InitExceptions;

exception Abort( Message: String );
exception Dump( Message: String );
exception XSegmentFault( S1,S2,S3,S4: Integer );  { segment fault }
exception XStackOverflow;                         { stack overflow }
exception DivZero;   { division by zero }
exception MulOvfl;   { overflow in multiplication }
exception StrIndx;   { string index out of range }
```

```
exception StrLong;    { string to be assigned is too long }
exception InxCase;    { array index or case expression out of range }
exception STLATETooDeep; { parameter in STLATE instruction is too large
exception UndfQcd;    { execution of an undefined Q-code }
exception UndfInt;    { undefined device interrupt detected }
exception IOSFlt;     { segment fault detected during I/O }
exception MParity;    { memory parity error }
exception EStack;     { E-stack wasn't empty at INCDDS }
exception OvflLI;     { Overflow in conversion to integer from Long Intege
exception OverReal;   { floating point overflow }
exception UndeReal;   { floating point underflow }
exception RealDiv0;   { floating point division by zero }
exception Real2Int;   { floating point real to integer overflow }

var ExcSeg: Integer;
```

```
procedure InitExceptions;
```

Abstract: InitExceptions tells the microcode what segment number to use when raising its own exceptions. The segment number is the one that the system assigns to this module.

Side affects: ExcSeg is set to the current segment number. The current segment is kept resident.

```
procedure RaiseP( ES, ER, PStart, PEnd: Integer );
```

Abstract: RaiseP is called to raise an exception. The compiler generates a call to RaiseP in response to

==========

raise SomeException(original parameters)

==========

in the following way

========== Push original parameters onto the MStack.

RAISE SegmentNumber(SomeException)
RoutineNumber(SomeException) ParameterSize

==========

The microcode calls RaiseP in the following way.

==========

Push parameters onto the MStack if appropriate.

ParameterSize := WordsOfParameters.

Error := ErrorNumber, Goto(CallRaise).

==========

Where CallRaise does the following.

==========

SaveTP := TP.

Push ExcSeg onto the MStack.

Push Error onto the MStack.

Push SaveTP-ParameterSize+1 onto the MStack.

Push SaveTP+1 onto the MStack.

call RaiseP.

==========

Parameters:
ER - Routine number of the exception to be raised.
ES - Segment number of the exception to be raised.
PStart- Pointer to the original parameters (as an offset from the base of the stack).
PEnd - Pointer to the first word after the original parameters (as an offset from the base of the stack).

Calls: Appropriate exception handler or HandleAll.

Design: See the "PERQ QCode Reference Manual, Q-Machine Architecture" for a description of the format of exception enable blocks and the format of variable routine descriptors.

RaiseP searches the exception enable list of each routine in the dynamic chain. When it finds one that matches ER and ES it searches the dynamic chain again to see if the specified handler is already active. If it is active, RaiseP continues searching the exception lists and dynamic chain where it left off. This is done in order to allow a handler to re-raise the same exception, and to prevent unlimited recursion in an exception handler that has a bug.

If an exception handler is found, the original parameters are pushed onto the MStack and the handler is called with CALLV.

If no exception handler is found, HandleAll is called.

*** RaiseP may not contain any exception handlers.

*** RaiseP must be guaranteed to be resident.

```
)dule FileAccess;


)stract:
    Module  to  handle  reading, writing, entering and deleting files
    independant from the directory structure.

    Written by the CMU Spice Group
)rsion Number V1.7
{*****************} exports {****************************}

)ports Arith from Arith;
)ports DiskIO from DiskIO;
)ports AllocDisk from AllocDisk;

)nction  CreateSpiceSegment(partition : integer; kind : SpiceSegKind) : SegID;
)ocedure DestroySpiceSegment(id : SegID);
)ocedure TruncateSpiceSegment(id : SegID; len : integer);
)ocedure ReadSpiceSegment(id : SegID; firstblk,numblks : integer;
                          ptr : ptrDiskBuffer);
)ocedure WriteSpiceSegment(id : SegID; firstblk,numblks : integer;
                           ptr : ptrDiskBuffer);
)ocedure Index(logblk : integer; var indblk,indoff : integer);

)ception BadLength(len: integer);

    Abstract: Raised if try to truncate file to a length < 0

    Parameters: len is bad length

)ception NotAFile(id: SegID);

    Abstract: Raised when an operation is  attempted  and  the  SegID
         passed does not
              seem to be the id for a valid file

    Parameters: id is the bad id
```

```
Procedure Index(logblk : integer; var indblk,indoff : integer);
```

Abstract: Find the index block and the offset from the top of the block for a logical block of a file

Parameters: logBlk - the logical block of the file to look up; may be negative
indBlk - the logical block number of the index block which holds the address for logblk
indoff - the offsetin indBlk to use in reading the address (the array index to use in DiskBuffer^.Addr). It is correctly set even if the indBlk is the FIBlk

```
function CreateSpiceSegment(partition : integer; kind : SpiceSegKind)
    : SegID;
```

Abstract: Create a new empty file on partition specified

Parameters: partition is the partition in which to allocate file; kind is the

Returns: ID of file created

Errors: Raises NotAFile if block at id does not seem to be a valid FIBlk

```
Procedure DestroySpiceSegment(id : SegID);
```

Abstract: Delete a file

Parameters: id is the SegId of file to delete

SideEffects: removes id from filesystem

Errors: Raises NotAFile if block at id does not seem to be a valid FIBlk

```
Procedure TruncateSpiceSegment(id : SegID; len : integer);
```

Abstract: Removes blocks from file to make the new length len

Parameters: id is the SegId of file; len is the new length (one greater than the last logical block number since files start at 0)

SideEffects: Shortens the file

Errors: Raises BadLength is length to truncate file to is < 0
Raises NotAFile if block at id does not seem to be a valid FIBlk

```
Procedure ReadSpiceSegment(id : SegID; firstblk,numblks : integer; ptr
     : ptrDiskBuffer);

     Abstract: Reads one or more blocks from file

     Parameters: id       - the SegId of file;
          firstBlk - the logical blk # of first to read
          numBlks  - the number of blocks to read
          ptr      -  where the data should be put NOTE: If the blocks
               specified to read don't  exist;  ptr^  is  filled  with
               zeros

     Errors:  Raises  NotAFile  if  block  at id does not seem to be a
          valid FIBlk


Procedure WriteSpiceSegment(id : SegID;  firstblk,numblks  :  integer;
     ptr : ptrDiskBuffer);

     Abstract: Writes one or more blocks onto file

     Parameters: id       - the SegId of file;
          firstBlk - the logical blk # of first to write
          numBlks  - the number of blocks to write
          ptr      - where the data should come from

     SideEffects:  Changes  the  data  in  the  file and may cause new
          blocks to be allocated and file length changed

     Errors: Raises NotAFile if block at id does  not  seem  to  be  a
          valid FIBlk
```

```
Module FileDefs;


Abstract:

     Defines some constants and types needed by various people so
     FileSystem doesn't need to import DiskIO in its export section

     Written by: Brad A. Myers 3-Mar-81

     Copyright (C) 1981 Three Rivers Computer Corporation
Version Number V1.2
EXPORTS

Imports GetTimeStamp from GetTimeStamp; {Using TimeStamp}

const
  DBLZERO             = nil;    {a two word 0}

type
  FSBit8              = 0..255;
  FSBit16             = integer;
  FSBit32             = ^integer; {will be a long when compiler knows about 'em}

Const DISKBUFSIZE  = 256;       {defined by hardware, 256 words per sec}

type SegID          = FSBit32;  {In SpiceSeg, the virtual address of the
                                   -1 block of a file}
     DiskAddr       = FSBit32;  {The virtual address of a DiskBlock}

     SimpleName     = string[25];   {only the filename in the directory}
     PathName       = string[100];  {full name of file with partition and dev}
     PartialPathName = string[80]; {file name including all directories}
     FSOpenType     = (FSNotOpen, FSOpenRead, FSOpenWrite, FSOpenExecute);
     FSDataEntry    = packed record
                        FileBlocks       : integer;  {Size of file in blocks}
                        FileBits         : 0..4096;  {Number of bits in last blk}
                        FileSparse       : Boolean;  {true if can be sparse}
                        FileOpenHow      : FSOpenType;  {howOpen}
                        FileCreateDate   : TimeStamp;
                        FileWriteDate    : TimeStamp;
                        FileAccessDate   : TimeStamp;
                        FileType         : integer;  {see FileType.pas}
                        FileRights       : integer;  {protection code}
                        FileOwner        : FSBit8;   {UserId of file owner}
                        FileGroup        : FSBit8;   {GroupId}
                        Filename         : PartialPathName;
                    end;
    ptrFSDataEntry = ^FSDataEntry;
```

```
)S D.6 Interface                                        Module FileDir
; Feb 82

)dule FileDir;


)stract:

    The  directory  structure  for  PERQ  FileSystem Written by:  CMU
    Spice Group
)rsion Number V2.6
)**********************}Exports{**********************************}

)ports FileDefs from FileDefs;

)nction  GetFileID(name : PathName) : SegID;
)nction  PutFileID(var name : PathName; id : SegID) : boolean;
)nction  DeleteFileID(name : PathName) : SegID;
)nction  GetDisk(var name : PathName; var partition : integer) : boolean;

)r
 DefaultPartitionName : SimpleName; {includes device name and ends in a ">" }
 DefaultDeviceName    : SimpleName; {ends in a colon}
```

```
Function GetDisk(var name : PathName; var partition : integer) :
    boolean;
```

Abstract: Given a name, remove the device and partition specification and find the partition number

Parameters: name - the full file name to parse; the device and partition are optional. The device and partition if there are removed from the name string;
partition - set to the partition specified or the default

Returns: False if specified device or partition malformed or not there

SideEffects: Mounts the partition if not already

Calls: FindPartition, MountPartition

```
Function GetFileID(name : PathName) : SegID;
```

Abstract: Find the SegID for name (does a lookUp)

Parameters: name - the full name (including all directories and optional device and partition) of the file to look up

Returns: The SegID of the file or DBLZERO if not there or mal-formed

Calls: ParseFilename, GetRootDirID, GetIDFromDir

```
Function PutFileID(var name : PathName; id : SegID) : boolean;
```

Abstract: enters name with SegID id into a directory

Parameters: name - the full name (including all directories and optional device and partition) of the file to enter; it is changed to remove all ">..>" and ">.>"s and remove the device (the name returned can be entered in the FileID block's FSData.Filename).
id - SegID of file;

Returns: True if file successfully entered; false if device, partition or a sub-directory is mal-formed NOTE: ***IT IS ILLEGAL TO CALL PutFileID FOR A NAME THAT IS ALREADY IN THE*** ***DIRECTORY BUT THIS IS ONLY SOMETIMES CAUGHT IF ATTEMPTED***

Calls: ParseFilename, GetRootDirID, GetIDFromDir, PutIDInDir

Function DeleteFileID(name : PathName) : SegID;

Abstract: Removes the directory entry for name

Parameters: name - the full name (including all directories and optional device and partition) of the file to remove from directory

Returns: SegID of file removed from Directory or DBLZERO if not there or part of name is mal-formed

Calls: ParseFilename, GetRootDirID, GetIDFromDir

```
module FileSystem;


Abstract:


     Spice Interim File System.

     Written by: Richard F.  Rashid  Date       : February  24,  1981
     Copyright (C) 1981 - Carnegie-Mellon University
Version Number V7.3
{********************} Exports {********************}

imports FileDefs from FileDefs;

const
  FSVersion      = '7.3';     { File system version number }
  BlksPerFile    =#077777;    { Max blocks in each file }
  FirstBlk       =0;          { Block number of the first data block }
                              { in a file }
  LastBlk        =#077776;    { Block number of the last data block }
                              { in a file.  }
  FIBlk          =-1;         { Block number of the File Information Block }
  BootLength     = 60 + 128;  { Size of the bootstrap area on disk--the }
                              { first n blocks on the disk.  the microcode }
                              { boot area is 60 blocks, the Pascal boot  }
                              { area is 128 blocks (32K).  }
  StartBlk       =BootLength;{ The block number of the FIBlk of the first}
                              { user file.  }
  SysFile        = -1;        { File ID of the system area on disk.  }
  SEARCHSIZELIST = 5;         { Max number of directories on search list.  }

type
  DirBlk= Record                        { Record for reading disk blocks }
           Case Integer Of
           2: (
                Buffer:Array[0..255] Of Integer
              );
           3: (
                ByteBuffer: Packed Array [0..511] of FSBit8
              )
           End;

  PDirBlk= ^DirBlk;

  FileID         = integer;
  BlkNumbers     = integer;

  SearchList     = array[1..SEARCHSIZELIST] of PathName;
  ptrSearchList = ^SearchList;

var
  FSDirPrefix:PathName;  {current default directory including device and par
  FSSysSearchList: SearchList;

function FSLookUp(FileName:PathName;Var BlkInFile,BitsInLBlk: Integer): FileID;
```

```
                    {uses current system search list}

 .ction FSLocalLookUp(FileName:PathName; Var BlkInFile,BitsInLBlk: Integer):
          FileID;    {doesn't use any search lists}

inction  FSSearch(var slist : SearchList; var FileName : PathName;
                  var BlkInFile, BitsInLBlk: integer) : FileID;
                      {uses specified search list instead of system one; is
                       var so no copying; changes FileName to be full filename
                       actually used}

inction  FSEnter(FileName:PathName): FileID;
·ocedure FSClose(UserFile:FileID; Blks,Bits:Integer);
·ocedure FSBlkRead(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
·ocedure FSBlkWrite(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
·ocedure FSInit;
·ocedure FSMount(disk : integer);
·ocedure FSDismount(disk : integer);

·ocedure FSSetPrefix(prefixname : PathName);  {FSSetPrefix just assigns the
                                                vble; use FileUtils.FSSetPath
                                                to do processing on new path}

·ocedure FSGetPrefix(var prefixname : PathName);
inction  FileIDtoSegID(id : FileID) : SegID;
inction  SegIDtoFileID(id : SegID) : FileID;
  ·edure FSSetupSystem(bootchar: integer);
  ·edure FixFilename(var filename : PathName; nulliserror : boolean);
inction FSIsFSDev(name: PathName; var devName: String): integer;

·ception FSNotFnd(name: PathName);

    Abstract: Raised if file looked up is not found. If this
         exception is not
              handled by client, the lookup or search will return
              zero

    Parameters: name is the name not found

·ception FSBadName(name: PathName);

    Abstract: Raised if file entered is illegal because:
              1) the device or partition specified is not valid 2) a
              directory name specified does not exist 3) the length
              of the simpleName is > 25 characters If this exception
              is not handled by the client, the Enter will return
              zero

    Parameters: name is the name that is illegal

inction FSInternalLookUp(FileName:PathName; Var BlkInFile,BitsInLBlk:Integer):
           FileID;

·ception FSDirClose;
```

Abstract: Raised if attempt to FSClose a directory file. This is usually
a bad idea since directories are spare files with an invalid length field.

RESUME: Allowed. Will close the file as if nothing had happened.

```
const
  FSDebug = false;
```

```
Function SegIDtoFileID(id : SegID) : FileID;
```

Abstract: Convert a two word SegId into a one word fileID

Parameters: id is a two word segID

Returns: A one word FileID; it may be pos or neg or zero

```
Function FileIDtoSegID(id : FileID) : SegID;
```

Abstract: Convert a one word FileID into a two word SegID

Parameters: id is a one word FileID

Returns: a two word SegID

```
Procedure FSInit;
```

Abstract: Initializes the FileSystem; call BEFORE FSSetUpSystem; Also initialize SegSystem and DirSystem

SideEffects: Initializes; sets global Initialized to true; sets Prefix and Search list to null

```
Procedure FixFilename(var filename : PathName; nulliserror : boolean);
```

Abstract: Makes fileName a full path name by adding as many defaults as necessary

Parameters: filename is name to fix; it is modified to have the full path name as follows:
(dev):(rest) - no change
:(rest) - adds DefaultDevice from AllocDisk to front
>(rest) - adds DefaultPartition from AllocDisk to front
(rest) - adds FSDirPrefix to front; if nullIsError-then no change to name if fileName = '' else changes '' to FSDirPrefix

Errors: allows STRLong to pass through from PERQ_String; This means that fileName is invalid

```
Procedure FSMount(disk: integer);
```

Abstract: Mounts the disk specified and prints all partitions

Parameters: Disk is device to mount (0=HardDisk, 1=Floppy)

Calls: DeviceMount and DisplayPartitions

```
Procedure FSDismount(disk: integer);
```

Abstract: Dismounts the disk specified and prints all partitions

Parameters: Disk is device to dismount (0=HardDisk, 1=Floppy)

Calls: DeviceDismount and DisplayPartitions

```
Procedure FSSetPrefix(prefixname : PathName);
```

Abstract: Sets the default pathName

Parameters: prefixname is new name; no checking is done

SideEffects: changes FSDirPrefix

```
Procedure FSGetPrefix(var prefixname : PathName);
```

Abstract: Returns the default pathName

Parameters: prefixname is current name; it is set with current value

```
Function FSInternalLookUp(FileName:PathName;Var
BlkInFile,BitsInLBlk:Integer): FileID;
```

Abstract: Does a lookup of FileName in the current path only.

Parameters: FileName is a filename. BlkInFile and BitsInLBlk are set with the number of blocks in the file and the number of bits in the last block respectively.

Returns: 0 if file doesn't exist; else the FileID of the file

Errors: This procedure does not raise any errors

SideEffects: Sets the FileAccessDate of the file

Calls: FixFileName, GetFileID, GetTStamp, SegIDToFileID

```
Function FSLocalLookUp(FileName:PathName;Var
BlkInFile,BitsInLBlk:Integer): FileID;
```

Abstract: Does a lookup of FileName in the current path only.

Parameters: FileName is a filename. BlkInFile and BitsInLBlk are set with the number of blocks in the file and the number of bits in the last block respectively.

Returns: 0 if file doesn't exist; else the FileID of the file

SideEffects: Sets the FileAccessDate of the file

Errors: Raises FSNotFnd if file not there (if not caught, then lookup returns 0)

Calls: InternalLookUp

Function FSSearch(var slist : SearchList; var filename : PathName; var blkinfile,bitsinlblk: integer) : FileID;

Abstract: Does a lookup of FileName straight first and then with each of the names in slist on the front.

Parameters: slist - a searchList; any non-'' entries are assumed to be paths and are put on the front of the filename. The first one to be tried is slist[1]. The first match is the one used; No checking is done on the validity of the entries in slist

filename - the file to be looked up; it is changed to be the full name of the file if found. If fileName is empty then file not found.

BlkInFile and BitsInLBlk - set with the number of blocks in the file and the number of bits in the last block respectively.

Returns: 0 if file doesn't exist in any path; else the FileID of the file

SideEffects: Sets the FileAccessDate of the file

Errors: Raises FSNotFnd if file not there (if not caught, then lookup returns 0)

Calls: FSLocalLookUp, Concat

Function FSLookUp(FileName: PathName; Var BlkInFile, BitsInLBlk: Integer) : FileID;

Abstract: Does a lookup of fileName first in the current path and then in each of the entries of the system search list.

Parameters: filename is the file to be looked up; BlkInFile and BitsInLBlk are set with the number of blocks in the file and the number of bits in the last block respectively.

Returns: 0 if file doesn't exist in any path; else the FileID of the file

SideEffects: Sets the FileAccessDate of the file

Errors: Raises FSNotFnd if file not there (if not caught, then lookup returns 0)

Calls: FSSearch with FSSysSearchList as the sList

```
Function FSEnter(FileName:PathName): FileID;
```

Abstract: Enters the file in the current path.

Parameters: filename is the file to be entered. It may or may not exist; if not exists then is created;

Returns: 0 if file can't be created because part of its name is invalid (e.g. the device, partition or directory specified doesn't exist) else the FileID of the file

SideEffects: Creates a file if necessary and enters it into the directory; if creating, then sets size to zero and create date; Sets type to 0 (UnknownFile); whether or not creating; sets WriteDate and AccessDate

Errors: Raises FSBadName if name passed is illegal due to a device, partition, or directory name in path not existing or target name is longer than 25 characters or illegal in some other way. If this exception is not caught, Enter returns zero.

```
Procedure FSClose(UserFile:FileID; Blks,Bits:Integer);
```

Abstract: Closes a file (setting size).

Parameters: UserFile is ID of file to close; Blks is the size of the file in blks and bits is the number of bits in the last block;

SideEffects: Truncates file to size specified; does a FlushAll

Errors: Raises FSDirClose if attempt to close a directory file. If resume from this exception, then closes normally.

```
Procedure FSBlkWrite(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
```

Abstract: Writes one block onto a file.

Parameters: UserFile is ID of file to write on Block is number of block to write (starting at zero); Buff is buffer holding data to write onto the file at Block

SideEffects: Changes the data of block Block

Calls: WriteSpiceSegment

```
Procedure FSBlkRead(UserFile:FileID; Block:BlkNumbers; Buff:PDirBlk);
```

Abstract: Reads one block of a file. If block specified is not part of the file then simply zeros the buffer

Parameters: UserFile is ID of file to read from Block is number of block to read (starting at zero); Buff is buffer to copy data into

Calls: ReadSpiceSegment

```
Procedure FSSetupSystem(bootchar: integer);
```

Abstract: Call this after FSInit to set up the system and print a lot of messages

Parameters: boot char is ord of key held down to boot

SideEffects: Mounts device from which booted; Mounts all of its partitions Sets AllocDisk's DefaultDeviceName and DefaultPartitionName. Sets FSDirPrefix to be root of current Partition and adds that path to the bottom of the search list

```
Function FSIsFSDev(name: PathName; var devName: String): integer;
```

Abstract: determine whether name is a file that the filesystem knows how to handle

Parameters: name is a name of a file; devName will be assigned the device name IF NOT FSDevice DevName will be in upper case and does NOT contain the colon.

Returns: 0 if name doesn't contain a : or if dev is one the filesystem knows about; the index of the colon otherwise

```
Module FileTypes;


Abstract:

     This module exports the Types put in the FileType field of File
     FIBs.  The types are stored as integers.  Three Rivers reserves
     the first 512 types for their use.  Customers are encouraged to
     choose numbers > 512 if they invent new file types

     Written by Brad A.  Myers  Feb.  2, 1981

     Copyright (C) 1980 Three Rivers Computer Corporation
Version Number V1.2
{\\\\\\\\\\\\\\\\\\\\\\\\\\} EXPORTS {//////////////////////////}

Const
      UnknownFile = 0;
      SegFile = 1;
      PasFile = 2;
      DirFile = 3;
      ExDirFile = 4;
      FontFile = 5;
      RunFile = 6;
      TextFile = 7;      {for non-Pas text files}
      CursorFile = 8;    {cursor bin files}
      BinaryFile = 9;
      BinFile = 10;      {microcode output}
      MicroFile = 11;
      ComFile = 12;
      RelFile = 13;
      IncludeFile = 14; {included in a pas file}
      SBootFile = 15; {system part of boot file}
      MBootFile = 16; {microcode part}
      SwapFile = 17; {a file used for swapping by compiler or editor; leng
                      not set}
      BadFile = 18;  {created by the scavenger}
```

```
module FileUtils;


     Filesystem utilities not needed by the system

     Written by Brad Myers.  March 5, 1981.

Version Number V1.10
{********************} Exports {********************}

imports FileSystem from FileSystem;

type
  ptrScanRecord = ^ScanRecord;
  ScanRecord    = record
                    InitialCall : boolean;
                    Blk         : DiskAddr;
                    Entry       : Integer;
                    DirName     : PathName;
                  end;

Procedure FSDelete(filename: PathName);
Function  FSScan(scanptr : ptrScanRecord; var name : SimpleName;
                 var id : FileID) : boolean;
Procedure FSRename(SrcName, DestName: PathName);
Function FSMakeDirectory(var DirName: PathName): FileID;
Procedure FSSetSearchList(sList: SearchList);
Procedure FSPopSearchItem(var sList: SearchList);
Procedure FSPushSearchItem(name: PathName; var sList: SearchList);
Procedure FSAddToTitleLine(msg: String); {adds as much of msg as possible to
                                          title line after the current path}
Exception DelError(FileName: PathName);

     Abstract: Raised when can't delete file (because not there)

     Parameters: FileName is file that can't delete

Exception RenError(msg: String; FileName: PathName);

     Abstract: Raised when can't rename file

     Parameters:  msg is reason can't rename and fileName is file with
          the problem.    To   print   message,   use    "WriteLn('**
           ',msg,filename);"

Exception MkDirErr(msg: String; dirName: PathName);

     Abstract: Raised when can't make a directory because
               1) a file named dirName already exists 2) dirName
               cannot be entered (bad subdir part) 3) dirName is empty
               4) dirName is ROOT.DR (reserved directory name)
```

```
    Parameters:  msg explains problem with makedir attempt;
                 dirName is name attempted  to  use.   Use  "WriteLn('**
                 ',msg,dirName);"

Exception SrchWarn(fileName: PathName);

    Abstract: Raised if try to Pop last item or push into  last  hole
        of the
             Search List

    Parameters: '' if Pop; name of item trying to push if Push

    Resume: ALLOWED; if resume then does the operation anyway

Exception SrchErr(fileName: PathName);

    Abstract: Raised if try to Pop empty list or push onto full  list
        for the
             Search List

    Parameters: '' if Pop; name of item trying to push if Push

    Resume: NOT allowed

Function FSExtSearch(var SList : SearchList; Extensions: String;
                     var FileName : PathName;
                     var BlksInFile, BitsInLBlkInLBlk: Integer) : FileID

Exception RenToExist(fileName: PathName);

    Abstract:  Raised  when  try  to  rename  to  a file that already
        exists.  Not
             raised if renaming a file to its own name (no-op).

    Parameters: fileName - new name that already exists

    Resume: ALLOWED; If you wish to rename anyway; just continue  and
        FSRename
             will  delete  the DestName; In this case; you should be
             prepared to accept DelError;

Exception RenDir(fileName: PathName);

    Abstract: Raised when try to rename a directory.

    Parameters: fileName - name of the source directory.

    Resume:  ALLOWED; If you wish to rename anyway; just continue and
        FSRename
             will do the operation.  RenToExist etc.  may  still  be
             raised.

Procedure FSGetFSData(id: FileID; pData: ptrFSDataEntry);
Procedure FSSetFSData(id: FileID; pData: ptrFSDataEntry);
Procedure FSRemoveDots(var fname: PathName);
```

```
Procedure FSDelete(filename : PathName);
```

Abstract: Deletes filename from directory and from filesystem; fileName is deleted from the current path only (not search lists) if it doesn't contain device or partition info

Parameters: filename is the name of the file to be deleted

SideEffects: filename is deleted from the current directory if it exists; if not then nothing is done (and the user is not notified)

Calls: DeleteFileID; DestroySpiceSegment

Errors: Raises DelError(fileName) if can't delete file

```
Procedure FSRename(SrcName, DestName: PathName);
```

Abstract: Changes the name of SrcName to DestName; both are in the current path (not search lists) if not fully specified

Parameters: SrcName is the name of the file to change and DestName is the name it should be given

Returns: True if rename is successful; false if can't be done because:
1) - destName already exists
2) - SrcName and destName are in different partitions
3) - SrcName doesn't exist
4) - SrcName or DestName is malformed

SideEffects: The name of the file corresponding to SrcName is changed

Calls: DeleteFileID; DestroySpiceSegment

Errors: Raises RenError(msg, fileName) - if can't rename file where message explains why (do "Write(' ** ',msg,fileName)" in handler) Raises RenToExist(DestName) - if filename already exists; If you wish to rename anyway; just continue and FSRename will delete the DestName; In this case; you should be prepared to accept DelError;

Function FSScan(scanptr : ptrScanRecord; var name: SimpleName; var id : FileID): boolean;

Abstract: At each call returns the next entry in a directory. The names returned are in random order.

Parameters: scanPtr is a pointer to a ScanRecord which controls the scan. At the first call, scanPtr^.InitialCall should be set to true and scanPtr^.dirName should be set to the directory to scan through. No fields should be modified by the caller after the initial setting. The dirName field of the scanPtr record is modified to contain the Full path name of the directory. name is set to the name of the file found on this call and id is its fileID; scanPtr is modified after each call so the next call will return the next name in the directory

Returns: True if a valid name and id returned; false if the directory has been exhausted in which case name and id are NOT valid

Function FSMakeDirectory(var dirName: PathName): FileID;

Abstract: Create a new directory named dirName.

Parameters: DirName is the name of the directory to create; the name is changed to be the full path name of the directory created

Returns: The fileID of the directory

SideEffects: Creates a file named dirName (appending a ".DR" to end if not there. Sets the FileType field to DirFile; and sets the FileBits to 4096

Errors: Raises MkDirErr(msg, dirName) if 1) a file named dirName already exists 2) dirName cannot be entered (bad subdir part) 3) dirName is empty 4) dirName is ROOT.DR (reserved directory name) where msg describes error. Do not continue from this signal

Procedure FSSetSearchList(sList: SearchList);

Abstract: Assign the system search list.

Parameters: sList is new search list. It is a bad idea to not include a partition which contains a full set of system files

SideEffects: Changes system search list

Procedure FSPopSearchItem(var sList: SearchList);

Abstract: Removes the most recent item from the search list

Parameters: sList is search list to pop from (it is modified)

Errors: Raises SrchWarn('') if try to pop last item; if continue from it then pops it anyway; Raises SrchErr('') if list empty and try to pop; don't continue from this one

Procedure FSPushSearchItem(name: PathName; var sList: SearchList);

Abstract: adds name to the front of the search list

Parameters: name is new name to add to the front of the search list searchList is modified to have name at front

Errors: Raises SrchWarn(name) if try to push into last item; if continue from it then pushes it anyway; Raises SrchErr(name) if list full and try to push; don't continue from this one

Environment: Assumes oldest item in list is at high position (e.g. 5)

Procedure FSAddToTitleLine(msg: String);

Abstract: adds as much of msg as possible to title line after the current path which is truncated to 35 characters

Parameters: msg is string to be displayed. The first 43 characters of it are displayed

Side Effects: Changes current window's title line

Procedure FSGetFSData(id: FileID; pData: ptrFSDataEntry);

Abstract: Returns the FSDataEntry description of a file

Parameters: id is the FileID for the file that data wanted for pData is a pointer to a data block to which the FSData is copied. Memory for this pointer must be allocated before the call

```
Procedure FSSetFSData(id: FileID; pData: ptrFSDataEntry);
```

Abstract: Changes the FSDataEntry of a file

Parameters: id is the fileID of the file to be modified pData is the FSDataEntry to set id to. The entire FSDataEntry description of id is changed, so the user should use FSGetFSData to read the FSDataEntry and then change the desired fields only

Side Effects: Changes the FSDataEntry for id

```
Function FSExtSearch(var SList : SearchList; Extensions: String; var
    FileName : PathName; var BlksInFile, BitsInLBlkInLBlk: Integer) :
    FileID;
```

Abstract: FSExtSearch performs a breadth-first lookup of a file using a specified searchlist and a list of extensions. The search order is as follows: 1) Try the name with each extension in the current directory. 2) Repeat steps 1 in each path specified in the searchlist. If the file is found, the FileName is changed to be the full file name actually found.

Parameters:

- SList - Searchlist to use.
- Extensions - List of extensions to try with a single space after each extension. E.g. '.Pas .Micro .Cmd .Dfs '. The string must have a single trailing space. A single leading space or a pair of adjacent spaces causes FSExtSearch to look for the file exactly as typed (with no extension appended). Other extra spaces are not allowed. If Extensions does not end in a space, then one added.
- FileName - Name of file to find, set to be the full name of the file that was actually found.
- BlksInFile - Length of file in blocks.
- BitsInLBlk - Bits in last block of file.

Returns: 0 if file not found or id of file

Errors: Raises FSNotFnd if file not found

```
Procedure FSRemoveDots(var fname: PathName);
```

Abstract: Removes "."s and ".."s from file name leaving correct full name

Parameters: fname is file name. It is changed to not have dots.

```
module GetTimeStamp;


GetTimeStamp - Perq get time routine.
J.  P.  Strait        1 Feb 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     GetTimeStamp implements the read-time-as-TimeStamp  function  for
     the Clock module.  See the Clock module for more details.

     Design:  GetTimeStamp  is  a  separate  module  so that it may be
     imported into the resident system without importing all the other
     Clock routines.  Once virtual memory is implemented, GetTimeStamp
     and Clock should be merged into a single module.

Version Number V1.4
{//////////////////////////} Exports {\\\\\\\\\\\\\\\\\\\\\\\\\}

const GetTSVersion = '1.4';


type TimeStamp = packed record
       { the fields in this record are ordered this way to optimize bits }
       Hour:   0..23;
       Day:    1..31;
       Second: 0..59;
       Minute: 0..59;
       Month:  1..12;
       Year:   0..63;  { year since 1980 }
       end;

     TimeReference = record
                       Lower: Integer;
                       Upper: Integer
                       end;

procedure GetTStamp( var Stamp: TimeStamp );


var PastStamp: TimeStamp;
    Past: TimeReference;
```

```
procedure GetTStamp( var Stamp: TimeStamp );
```

Abstract: returns a timeStamp for the current time

Parameters: Stamp is set to be the stamp for the current time

```
module gpib;


Abstract:
     Support  routines for PERQ GPIB devices.  The package maintains a
     buffer (gpCommandBuffer) which holds either data bytes (sent with
     procedure  gpPutByte)  or  Auxiliary  commands  (sent  with
     gpAuxCommand).  The buffer is sent to the 9914 when full, or when
     gpFlushBuffer  is  called.  If  the  buffdr  has data bytes when
     gpAuxCommand is called, it will do a  gpFlushBuffer.  Similarly,
     when  gpPutByte is called, it will flush the buffer, if auxiliary
     commands are in gpCommandBuffer.

     writen by Brian Rosen
     Copyright(C) 1980,  Three  Rivers  Computer Corporation

Version Number V1.3

exports
 const GpibVersion = '1.3';
       gpBufSize = 32;
       gpBufMax = 31; {gpBufSize - 1}
 { the following codes are the IEE488-1975 Controller Command Codes
   they are issued by the Controller-In-Charge while asserting ATN }
       gpacg = #000; {addressed group command}
       gpdcl = #024; {device clear}
       gpget = #010; {group execute trigger}
       gpgtl = #001; {go to local}
       gplag = #040; {listen address group}
       gpllo = #021; {local lockout}
       gpmla = #040; {my listen address}
       gpmta = #100; {my talk address}
       gpmsa = #140; {my secondary address}
       gpppc = #005; {parallel poll configure}
       gpppe = #140; {parallel poll enable}
       gpppd = #160; {parallel poll disable}
       gpppu = #025; {parallel poll unconfigure}
       gpscg = #140; {secondary copmmand group}
       gpsdc = #004; {selected device clear}
       gpspd = #061; {serial poll disable}
       gpspe = #060; {serial poll enable}
       gptct = #011; {take control}
       gptag = #100; {tahk address group}
       gpuag = #020; {universal address group}
       gpunl = #077; {unlisten}
       gpunt = #137; {untalk}
 type  { these commands are the major state change control commands
         of the TMS9914 chip which forms the interface to the GPIB
         Consult the TI documentation on the TMS9914 for more information}

       {These definitions are order dependent}
       gpAuxiliaryCommands = (gpswrst,   {Chip Reset}
                              gpdacr,    {Release DAC holdoff}
                              gprhdf,    {Release RFD holdoff}
                              gphdfa,    {Holdoff all data}
```

```
                                gphdfe,     {Holdoff on End}
                                gpnbaf,     {Set NewByteAVailable false}
                                gpfget,     {Force Group Execute Trigger}
                                gprtl,      {Return to Local}
                                gpfeoi,     {force End or Identify}
                                gplon,      {Listen Only}
                                gpton,      {Talk Only}
                                gpgts,      {GoTo Standby}
                                gptca,      {Take Control Asynchronously}
                                gptcs,      {Take Control Synchronously}
                                gprpp,      {Request Parallel Poll}
                                gpsic,      {Set Interface Clear}
                                gpsre,      {Set Remote Enable}
                                gprqc,      {Request Control}
                                gprlc,      {Release Control}
                                gpdai,      {Disable All Interrupts}
                                gppts,      {Pass Through next Secondary}
                                gpstdl,     {Set T1 Delay}
                                gpshdw);    {Shadow Handshake}
        gpParmType = (gpOff, gpOn, gpDontCare);  {parameters for Aux Commai
        gpByte = 0..255;    {Data byte for gpib transactions}
        gpRange = 0..gpBufSize;
        gpDeviceAddress = 0..31;          {legal addresses for devices on G
        gpBuffer = packed array [gpRange] of gpByte;
        gppBuffer = ^gpBuffer;
 var    gpCommandBuffer: gppBuffer;       {place to put commands}
        gpBufPtr: 0..gpBufSize;           {pointer to gpCommandBuffer}
        gpHaveDataBytes, gpHaveAuxiliaryCommands: boolean;
                        {true if buffer in use}

{   The package maintains a buffer (gpCommandBuffer) which holds
    either Data bytes (sent with proceedure gpPutByte)
    or Auxialiary Commands (sent with gpAuxCommand)
    The buffer is sent to the 9914 when full, or when ghForceBuffer
    is called.  If the buffer has data bytes when gpAuxilairyCommand is
    called, it will do a gpForceBuffer.  Similarly, when gpPutByte
    is called, it will force the buffer if auxiliary commands are in
    gpCommandBuffer
}

          { Initialze GPIB package, called once only, turns off tablet }
procedure gpInit;


          { Send an auxiliary command to TMS9914
            some commands require a parameter (gpOff/gpOn) }
procedure gpAuxCommand(gpCmd: gpAuxiliaryCommands; gpParm:gpParmType);

          { Put a data byte or a Control byte out on the data bus
            TMS9914 must be in Controller Actives State if the byte is a
            controller command byte.  Must be in Talk Only if a data byte
procedure gpPutByte(gpData: integer);

          { Sends all bytes in buffer }
procedure gpFlushBuffer;

          { Set TMS9914 to be a Talker, set a device to be a listener
```

```
            This procedure takes control of the bus, unlistens and untalks
            all devices (including itself), and sets a listener with
            MyListenAddress then sets TMS9914 to be the talker with TalkONly }
·ocedure gpITalkHeListens(gpAddr: gpDeviceAddress);

          { Set TMS9914 to be a Listener, set a device to be a talker
            This procedure takes control of the bus, unlistens and untalks
            all devices (including itself), and sets a talker with
            MyTalkAddress then sets TMS9914 to be the listener with ListenONly}
·ocedure gpHeTalksIListen(gpAddr: gpDeviceAddress);

          { turn the BitPad (device addres #10) off }
·ocedure gpTbltOn;
          { turn the BitPad back on again }
·ocedure gpTbltOff;
          { Send a buffer of user data to the 9914 }
·ocedure gpSend(var gpBuf: gppBuffer; gpCount: gpRange);
          {Get a buffer of data from the 9914 (Not implemented yet) }
·ocedure gpReceive(var gpBuf: gppBuffer; gpCount: gpRange);
          { Get a byte of data from the GPIB }
ınction  gpGetByte: gpByte;

(ception GPIBerror(SoftStatus: integer);
```

Abstract:

Raised when GPIB encounters an error indication in softstatus from UnitIO or IOCRead. The condition should be corrected and the operation retried. The most likely error is a timeout: IOETIM (See IOErrors).

```
procedure gpInit;
```

```
Module Helper;


WJHansen        Jan 82.
Copyright (C) Three Rivers Computer Corporation, 1982.

Abstract:
     Reads  an  index  file and presents options for assistance to the
     user.
Version Number V1.1
exports

    imports FileDefs from FileDefs;        {for PathName}

procedure GiveHelp(FName:PathName);
```

```
procedure GiveHelp(FName:PathName);
```

Abstract: Reads a help index and displays it. Lets user ask for information on topics in the index and displays the files containing those topics.

Parameters:

FName - Name of the file containing the index. The path to this file is used as the path to the individual help files.

Module IO;

Abstract:

PERQ Raw IO Drivers - Compatibility file.
Written by: Miles A. Barel

The IO module has been split into four modules: IO_Init, IO>Unit, IO_Others, and IO_Private. This module is provided for compatibility. It imports the first three modules in its exports section. IO_Private is not exported because it provides definitions which were private in the old IO module and is used only by the new IO modules.

Design: 1) Interrupt routines must *never* cause segment faults. 2) UnitIO must increment and decrement the IOCount of the segments which are involved in IO. 3) Segment faults must *never* happen while interrupts are off.

```
{*******************************}   Exports   {*************************
```

```
Const IOVersion = '4.8';
```

```
Imports IO_Init from IO_Init;
Imports IO_Unit from IO_Unit;
Imports IO_Others from IO_Others;
```

```
Module IOErrMessages;

Abstract:

     This  module exports a procedure to return an  error string for a
     disk error

     Written by : Brad A.  Myers  May 12, 1981 Copyright  (C)  1981  -
     Three Rivers Computer Corporation
Version Number V1.1
{//////////////////////////} EXPORTS {\\\\\\\\\\\\\\\\\\\\\\\\}

Function IOErrString(err: integer): String;
```

```
Function IOErrString(err: integer): String;
```

Abstract: Returns a string describing a the error number

Parameters: err is the error number returned by UnitIO

Returns: A string describing the error

```
Module IOErrors;


Abstract:

     I/O System Error Code Definitions

     Copyright (C) 1981,1982 - The Three Rivers Computer Corporation
Version Number V1.2
Exports

Imports SystemDefs from SystemDefs;  {using Ether3MBaud}

Const
        IOEIOC = 1;         { IO Complete }
        IOEIOB = 0;         { IO Busy }
        IOEBUN = -1;        { Bad Unit Number }
        IOENBD = -2;        { Raw Block IO to this device is not implemented }
        IOEWRF = -3;        { Write Failure }
        IOEBSE = -4;        { BlockSize Error }
        IOEILC = -5;        { Illegal Command for this device }
        IOENHP = -6;        { Nil Header Pointer }
        IOEADR = -7;        { Address Error }
        IOEPHC = -8;        { Physical Header CRC Error }
        IOELHC = -9;        { Logical Header CRC Error }
        IOEDAC = -10;       { Data CRC Error }
        IOEDNI = -11;       { Device Not Idle }
        IOEUDE = -12;       { Undefined Error! }
        IOENCD = -13;       { Device is not a character device }
        IOECBF = -14;       { Circular Buffer Full }
        IOELHS = -15;       { Logical Header SerialNum Mismatch }
        IOELHB = -16;       { Logical Header Logical Block Number Mismatch }
        IOECOR = -17;       { Cylinder Out of Range }
        IOEDNR = -18;       { Device not ready }
        IOEMDA = -19;       { Missing data address mark }
        IOEMHA = -20;       { Missing header address mark }
        IOEDNW = -21;       { Device not writable }
        IOECMM = -22;       { Cylinder mis-match }
        IOESNF = -23;       { Sector not found }
        IOEOVR = -24;       { Overrun }
        IOEUEF = -25;       { Undetermined equipment fault }
        IOESOR = -26;       { Sector out of range }
        IOETIM = -27;       { Time out error }
        IOEFRS = -28;       { Floppy recalibrate done }
        IOEDRS = -29;       { Disk recalibrate done }
        IOETO  = -30;       { Can't find track zero }
{$ifc Ether3MBaud then}
        IOEPTL = -31;       { Ether3 - received packet too large }

    IOEFirstError = -31;
{$elsec}
    IOEFirstError = -30;
{$endc}

    IOELastError = 0;
```

```
Module IO_Init;


IO_Init - Initialize the IO system.
Miles A. Barel ca. 1 Jan 80.
Copyright (C) 1980, Three Rivers Computer Corporation

Abstract:
     IO_Init initializes the Interrupt Vector Table, the Device Table
     and associated buffers, the Screen Package, the tablet and
     cursor, and the Z80.

Version Number V5.9
{******************}  Exports  {**********************}



Procedure InitIO;
```

```
Procedure InitIO;
```

Abstract: InitIO initializes the Interrupt Vector Table, the Device Table and associated buffers, the Screen Package, the tablet and cursor, and the Z80.

```
Module IO_Others;

IO_Others - Miscellaneous IO routines.
Miles A. Barel ca. 1 Jan 80.
Copyright (C) 1980, Three Rivers Computer Corporation

Abstract:
     IO_Others exports routines for the Cursor, Table, Screen, Time,
     and Keyboard.

Version Number V5.7
{**********************}  Exports  {*************************}

Imports SystemDefs from SystemDefs;

{ tablet/cursor procedures }
Type     CursFunction = (CTWhite, CTCursorOnly, CTBlackHole, CTInvBlackHole
                         CTNormal, CTInvert, CTCursCompl, CTInvCursCompl);
         TabletMode = (relTablet, scrAbsTablet, tabAbsTablet, offTablet);
         CursMode = (OffCursor, TrackCursor, IndepCursor);
         CursorPattern = array[0..63,0..3] of integer;
         CurPatPtr = ^CursorPattern;

Var      TabRelX, TabRelY: integer;       { tablet relative coordinates }
         TabAbsX, TabAbsY: integer;       { tablet absolute coordinates }
         TabFinger: boolean;              { finger on tablet }
         TabSwitch: boolean;              { switch pushed down }
         TabWhite   : boolean;          { True if white button down }
         TabGreen   : boolean;          { True if green button down }
         TabBlue    : boolean;          { True if blue button down }
         TabYellow  : boolean;          { True if yellow button down }
         TabMouse   : integer;          { Actual output from mouse }
         DefaultCursor: CurPatPtr;        { default cursor pattern }

Procedure IOLoadCursor(Pat: CurPatPtr; pX, pY: integer);
                       { load user cursor pattern }

Procedure IOReadTablet(var tabX, tabY: integer);
                       { read tablet coordinates }

Procedure IOSetFunction(f: CursFunction);

Procedure IOSetModeTablet(m: TabletMode);
                      { set the mode to tell what kind of
                      tablet is currently in use }

Procedure IOCursorMode (m: CursMode);
                       { if track is true, then Tablet
                         coordinates are copied every 1/60th
                         second into the cursor position.  if
                         indep, then coordinates are changed
                         only by user.  If off, then no
                         cursor displayed }
```

```
Procedure IOSetCursorPos(x,y: Integer);
                         { if trackCursor is false, then sets
                          cursor x and y pos.  If tracking, then
                          sets both tablet and cursor.  }

Procedure IOSetTabPos (x,y: Integer);
                         { if trackCursor is false, then sets
                           tablet x and y pos.  If tracking, then
                           sets both tablet and cursor }

Procedure IOReadCursPicture(pat: CurPatPtr; var px, py: integer);
                         { copies current cursor picture into
                           pat and sets px and py with the
                           offsets for the current cursor }

Procedure IOGetTime(var t: double);  { Get the double word 60 Hertz time }

{Procedure to change screen size}
Procedure IOScreenSize(newSize: integer; complement: Boolean);
                         { newSize is number of scan lines in
                           new screen; must be a multiple of
                           128.  Complement tells whether the
                           rest of the screen should be the
                           opposite color from the displayed
                           part }

{ disable/enable keyboard interrupts }
Procedure IOKeyDisable( var OldKeyEnable: Boolean );

Procedure IOKeyEnable( OldKeyEnable: Boolean );

{ clear the IO type-ahead buffer }
Procedure IOKeyClear;
```

```
procedure IOCursorMode( M: CursMode );
```

Abstract:

Sets the mode for the cursor. If the mode m is set to TrackCursor, Tablet coordinates are copied every 1/60th second into the cursor position. If it's set to IndepCursor, coordinates are changed only by the user. If it is set to OffCursor, no cursor is displayed.

Parameters:

m - the new mode for the cursor.

```
procedure IOSetModeTablet( M: TabletMode );
```

Abstract:

Sets the mode to tell what kind of tablet is currently in use.

Parameters:

m - the mode for the tablet.

```
procedure IOLoadCursor( Pat: CurPatPtr; pX, pY: integer );
```

Abstract:

Loads a user cursor pattern into the screen cursor.

Parameters:

Pat - a pointer to a cursor. It should be quad-word aligned.

pX and pY - offsets in the cursor where the origin is thought to be. For example, if the cursor is a bull's eye, 31 bits diameter flushed to the upper left corner of the cursor box, using (pX, pY) = (15, 15) will have the cursor surround the things pointed at.

NOTE: This procedure supports a cursor which is 56 x 64; with a scan line length of 4

```
procedure IOReadCursPicture( Pat: CurPatPtr; var pX, pY: integer );
```

Abstract:

Copies the current cursor picture into Pat and sets pX and pY with the offsets for the current cursor.

Parameters:

Pat, pX, and pY are filled with data on the current cursor. Note that Pat must be quad-word aligned.

```
Procedure IOSetFunction( f: CursFunction );
```

Abstract:

Sets the cursor function.

Parameters:

f - the function to set the cursor to.

```
procedure IOSetCursorPos( x, y: integer );
```

Abstract:

If the cursor's mode is not TrackCursor, this procedure sets the cursor's x and y positions. If tracking, it sets both tablet and cursor.

Parameters:

x and y - the new cursor coordinates.

```
procedure IOSetTabPos( x, y: integer );
```

Abstract:

If the cursor's mode is not set to TrackCursor, IOSetTabPos sets the tablet's x and y positions. If the mode is TrackCursor, both tablet and cursor are set.

Parameters:

x and y - the new tablet coordinates.

```
procedure IOReadTablet( var tabX, tabY: integer );
```

Abstract:

Reads tablet coordinates.

Parameters:

tabX and tabY - set to x and y values of the tablet.

```
Procedure IOScreenSize( newSize: Integer; complemented: Boolean );
```

Abstract:

Changes the amount of screen visible to the user (the rest is turned off, hence not displayed). The cursor is prevented from going into the undisplayed part of the screen.

Parameters:

newSize - number of scan lines in new screen; it must be a multiple of 128.

Complement - tells whether the rest of the screen should be the opposite color of the displayed part.

```
Procedure IOGetTime( var t : double );
```

Abstract:

Reads the 60 Hertz clock.

Parameters:

t - set to the new time.

```
procedure IOKeyDisable( var OldKeyEnable: Boolean );
```

Abstract: IOKeyDisable is used to disable keyboard interrupts. This is used to delay processing of control-c, control-shift-c and control-shift-d at critical times. The old value of the keyboard interrupt enable is returned and must be passed back to IOKeyEnable when re-enabling keyboard interrupts. Characters typed while keyboard interrupts are disabled are remembered. When keyboard interrupts are re-enabled, the characters are processed.

Parameters:
OldKeyEnable - set to the old value of the enable.

```
procedure IOKeyEnable( OldKeyEnable: Boolean );
```

Abstract: IOKeyEnable is used to enable keyboard interrupts. The old value of the keyboard interrupt enable (as returned from IOKeyDisable) must be passed to IOKeyEnable when re-enabling keyboard interrupts. If characters were typed while keyboard interrupts were enabled, IOKeyEnable calls KeyIntr to process those characters. The master interrupt control (INTON and INTOFF QCodes) must be on when this procedure is called.

Parameters:
OldKeyEnable - the old value of the enable.

```
procedure IOKeyClear;
```

Abstract: IOKeyClear clears the keyboard type-ahead buffer.

```
Module IO_Private;


IO_Private - IO system private definitions and interrupt routines.
Miles A. Barel ca. 1 Jan 80.
Copyright (C) 1980, Three Rivers Computer Corporation

Abstract:
     IO_Private exports interrupt routines and definitions which are
     private to the modules which make up the IO system.

     Design: Interrupt routines must *never* cause segment faults.

Version Number V5.9
{*******************************} Exports {***************************



Imports SystemDefs from SystemDefs;
Imports IO_Unit from IO_Unit;
Imports IO_Others from IO_Others;
Imports Raster from Raster;


Const

        CirBufSize = #100-3;     { size of circular buffers - made so that
                                 { total size of CirularBuffer = #100 }

        DskBlockSize = 512;      { block sizes for fixed block size devices
        FlpBlockSize = 128;      { Size is in BYTES }
        SpkBlockSize = 128;
        TabBlockSize = 8;
        Z80BlockSize = 20;
        GPIBBlockSize = 1;
        Last12Sector = 202 * 4 * 30 -1; {Tracks * Heads * Sectors.  0 star

        DskPriority = 0;        { interrupt priorities }
        FlpPriority = 3;
        SpkPriority = 2;
{$ifc Ether3MBaud then}
        Ether3Priority = 12;
{$elsec}
{$ifc Ether10MBaud then}
        Ether10Priority = 12;
{$endc}
{$endc}
        GPIBInPriority = 11;
        GPIBOutPriority = 1;
        TabPriority = 10;
        KeyPriority = 5;
        RSIPriority = 6;
        RSOPriority = 7;
        Z80Priority = 8;
```

```
        PutPriority = 9;
        GetPriority = 4;

        PSFloppy = #14;             { Put Status Special Codes }
        PSGPIB = 0;
        PSZ80Monitor = #11;
        PSTablet = 6;
        PSKeyBoard = 7;
        PSRS232 = 5;
        PSClock = #12;

        GSFloppy = #40;             { Get Status Special Codes }
        GSGPIB = #100;
        GSZ80Monitor = #10;
        GSTablet = #2;
        GSKeyBoard = #4;
        GSRS232 = #1;
        GSClock = #20;

        DisCst0 = #1154;      { consts to terminate Display List; add in funct }
        DisCst1 = #1351;
        MinCurY = 0;            { Minimum Y value for cursor }
        MaxCurY = 1023;         { Maximum Y value for cursor }
        SegSize = 128;          { # lines / display segment }
        CrsHeight = 64;         { Height of the cursor }
        CrsConst0 = #370;       { constants to compute funny X position }
        VisOnly = #2000;        { mode bits - Visual screen only }
        VisAndCur = #2400;      { visual screen and cursor }
        Map = #100000;          { cursor map function }


        Ctrl C = chr(#3);
        Ctrl S = chr(#23);
        Ctrl Q = chr(#21);
        BlamCh = Chr(#303);       { untranslated shift-control-C }
        DumpCh = Chr(#304);       { untranslated shift-control-D }

ifc Ether3MBaud then}
        E3TIntEnable = #4;      { Ether3 constants }
        E3TDone = #2000;
        E3TError = #4000;
        E3TGo = #10;
        E3RIntEnable = #1;
        E3RDone = #400;
        E3RPromiscuous = #20;
        E3RError = #1000;
        E3RGo = #2;
        E3XmtMask = E3TDone + E3TError + E3TGo;
        E3XmtSucc = E3TDone + E3TGo;
        E3RecMask = E3RDone + E3RError + E3RGo;
        E3RecSucc = E3RDone + E3RGo;

        PackBuffLen = 511;
        E3RecCount = PackBuffLen - 4;
        MAXE3RECERRS = 20;
endc}
```

```
Type
        IOPtrKludge = record case integer of
                        1: (Buffer: IOBufPtr);
                        2: (Offset: Integer;
                            Segment: Integer)
                        end;

{$ifc Ether3MBaud then}
        EtherBuff = array [0..PackBuffLen] of integer;
        pEtherBuff = ^EtherBuff;
{$endc}

        CirBufPtr = ^ CircularBuffer;
        CBType = (KDBType, RSIType, RSOType, StdType);{ types of circular bufs
        CirBufItem = packed record      { what we put in circular buffers }
                ch: char;               { the character }
                case CBType of          { and device specific condition bits }
                    KDBType: (KDBUnused: 0..63;
                              KDBOverrun: boolean;      { true = overrun }
                              KDBError: boolean);       { OR of all error bits
                    RSIType: (RSIUnused: 0..7;
                              RSIBreak: boolean;        { true=break received }
                              RSIModem: boolean;        { true=modem change }
                              RSIParErr: boolean;       { true=parity error }
                              RSIOverrun: boolean;      { true=overrun error }
                              RSIError: boolean);       { OR of all error bit
                    RSOType: (RSOUnused: 0..255);
                    StdType: (StdHiByte: 0..255)
                end;


        CircularBuffer = packed record  { circular buffer used for character }
                                        { devices, SpGetCir and SpPutCir }
                Length: integer;        { # chars in the buffer }
                RdPtr: integer;         { where to get chrs out }
                WrPtr: integer;         { where to put them in }
                Buffer: packed array[0..CirBufSize-1] of CirBufItem
                                        { last, the buffer }
            end;

        TabBufPtr = ^TabletBuffer;
        TabletBuffer = packed record    { buffer used for tablet, clock and }
                TabX: 0..#77777;        { Z80 Monitor Info }
                TabSwitch: boolean;
                TabY: 0..#77777;
                Fill: 0..1;
                ClkTime: double;
                Z80Mon: Z80Readings
            end;

        DispPtr = ^DisplayFile;
        DisplayFile = array[0..11] of
                packed record case boolean of
                    true: (int: integer);
                    false:(LineCount: 0..127;
```

```
                        StartOver: boolean;
                        ShowCursor: boolean;
                        VerticalRetrace: boolean;
                        ShowScreen: boolean;
                        DisableMicroInterrupt: boolean;
                        WriteBadParity: boolean;
                        Map: (CursOnly, CCursOnly, Compl, ComplInv,
                              Normal, Invert, CursComp, InvCursCompl))
            end;

    ScrCtlPtr = ^ ScrCtlBlock;
    ScrCtlBlock = packed record
            Cmd: DispPtr;
            ScreenBase: Integer;
            CursorBase: Integer;
            Unused1: Integer;
            Unused2: Integer;
            CursX: integer;
            filler: integer
        end;

)nst
    DskSPC = 30;            { Sectors Per Cynlinder }
    DskHds = 8;             { Max number of dsk heads }
    DskExHds = 0;           { Extra heads not in use }
    DskCyls = 202;          { Number of cylinders }
    FlpUnits = 4;           { 0 is the only valid unit }
    FlpSPC = 26;            { numbered 1 to 26 }
    FlpHds = 2;             { numbered 0 to 1 }
    FlpCyls = 77;           { numbered 0 to 77, 0 should not be used }

    TabAverage = 4;         { number of tablet points to average }
                            { this MUST be 4 }
    TabIgnore = 2;          { number of points to ignore when finger is }
                            { picked up or put down }
    TabFifoLen = TabAverage + TabIgnore;
    TabFifoMax = TabFifoLen - 1;
    GPIBxFudge = 38;        { actual range in X and Y for BitPad: 0..2200 }
    GPIByFudge = 1061;      { of TX and TY: 0..1100 }
                            { of TabAbsX: 0..1100 }
                            { of TabAbsY: 0..1100 }
                            { of TabRelX: -38..1062   limited to 0..767 }
                            { of TabRelY: 1061..-39   limited to 1023..0 }
    STopY = 0;
    SLeftX = 0;
    SRightX = 767;


/pe
    DskCmds = (DskIdle, DskRdCheck, DskDiagRead, DskWrCheck,
               DskWrFirst, DskFormat, DskSeek, DskClear);

  'c Ether3MBaud then}
    Ether3Cmds = (E3Rset, E3Status, E3Receive,
                  E3PromiscReceive, E3Transmit);
$endc}
```

```
        FlpCmds = (FlpUnUsed, FlpRead, FlpWrite, FlpFormat, FlpSeek, FlpReset

        GPIBCmds = (GPIBNop, GPIBWrite, GPIBWrEOI, GPIBCntl);

                    { Types of block devices }
        IOCBType = (DskType, FlpType, SpkType, GPIBType,
{$ifc Ether3MBaud then}
                                                    Ether3Type,
{$endc}
                    GenType);


        IOCBPtr = ^IOCB;
        IOCB = packed record { IOCBs must be 8 word aligned }
                Buffer: IOBufPtr; { data buffer for transaction }
                case IOCBType of
                    DskType:
                      ( DskCommand: 0..255;
                        DskNumSect: 0..255;
                       (* NEW FS [*)
                        DskAddr: Integer;
                       (* NEW FS *)
                       (* OLD FS [)
                        DskSector: 0..DskSPC-1;
                        DskHead: 0..DskHds+DskExHds-1;
                        DskCylinder: 0..DskCyls-1;
                       (* OLD FS *)
                        DskSerialNum: double;
                        DskLogBlk: integer;
                        DskFill1: integer;
                        DskNextAdr: double;
                        DskPrevAdr: double;
                        DskCntlError: ( OK,
                                        AddrsErr,       { address error }
                                        PHCRC,          { Physical Header CRC }
                                        LHSer,          { Logical Serial Wrong
                                        LHLB,           { Logical Block Wrong }
                                        LHCRC,          { Logical Header CRC }
                                        DaCRC,          { Data CRC }
                                        Busy);
                        DskFill2: boolean;
                        DskTrackZero: boolean;
                        DskWriteFault: boolean;
                        DskSeekComplete: boolean;
                        DskDriveRead: boolean;
                        DskNextIOCB: IOCBPtr);
                    FlpType:
                      ( FlpUnit: 0..FlpUnits-1;
                        FlpHead: 0..FlpHds-1;
                        FlpFill1: 0..31;
                        FlpCylinder: 0..255 { 0..FlpCyls-1 };
                        FlpSector: 0..255 { 1..FlpSPC };
                        FlpCommand: 0..255 { FlpCmds };
                        FlpByteCnt: integer { 0..256 };
                        FlpFill2: array [4..11] of integer;
```

```
                    FlpResult: integer;     { Not yet defined }
                    FlpNextIOCB: IOCBPtr);
                SpkType:
                  ( SpkFill0: 0..255;
                    SpkNumBufs: 0..255;
                    SpkFill1: array[3..11] of integer;
                    SpkFill2: 0..16383;
                    SpkAddrErr: boolean;
                    SpkError: boolean;
                    SpkNextIOCB: IOCBPtr);
                GPIBType:
                   (GPIBCommand: GPIBCmds;
                    GPIBF0: 0..63;
                    GPIBNumBufs: 0..255;
                    GPIBByteCount: 0..255;
                    GPIBF1: array [4..11] of integer;
                    GPIBResult: integer; {Not Implemented}
                    GPIBNextIOCB: IOCBPtr);
ifc Ether3MBaud then}
                Ether3Type:
                  ( Ether3Cmd: integer;
                    Ether3Delay: integer;
                    Ether3WdCnt: integer;
                    Ether3Status: integer;
                    Ether3NextIOCB: IOCBPtr );
endc}
                GenType:    { General Purpose entry }
                  ( GenCmd: 0..255;
                    NumBlks: 0..255;
                    GenFill0: array[3..11] of integer;
                    Result: integer;
                    NextIOCB: IOCBPtr)
        end { IOCB };

r
    KRBuf,                          { Keyboard Raw Buffer }
    KTBuf,                          { Keyboard translated buffer }
    RSIBuf,                         { RS-232 In Buffer }
    RSOBuf: CirBufPtr;              { RS-232 Out Buffer }
    TabBuf: TabBufPtr;              { Tablet/Clock Buffer }
    ScrBuf: ScrCtlPtr;              { Screen control blocks }
    DisFile0,DisFile1: DispPtr;     { Screen Display lists - double bufrs }
    OldCurY,                        { previous Cursor Y position }
    OldCurX: integer;               { previous Cursor X position MOD 8 }
    Cursor: CurPatPtr;              { Cursor Pattern }
    CursorX, CursorY: integer;      { new cursor coordinates }
    PointX, PointY: integer;        { the point of the cursor }
    TabFifo: array[0..TabFifoMax] of
             record X,Y: integer end;  { fifo of tablet points }
    TabFifoInx: integer;            { index into tablet fifo }
    TabCount: integer;              { counter for ignoring tablet points }
    SumX, SumY: integer;            { sum of 4 points for averaging }
    FlpLastCylinder: integer;       { last cylinder referenced }
    FlpLastHead: integer;           { last head referenced }
    GPIBTabletState: integer;       { GPIB tablet current state }
    GPIBxTablet, GPIByTablet: integer;  { GPIB tablet coordinates }
```

```
        GPIBInBuf: CirBufPtr;           { GPIB input buffer }
        StanleyTablet: boolean;         { if Stanley tablet is enabled }
        CurDskHds: integer;             { number of heads on this disk.  4 or
        CursF: integer;                 { function currently in use}
        BotCursF: integer;              { function for area below used area}
        BotComplemented: boolean;       { whether bot is complemented or not}
        SBottomY: integer;              { bottom of displayed area }
        TabMode: TabletMode;            { Current mode of the tablet }
        CCursMode: CursMode;            { Current mode of cursor }
        newFunct: Boolean;              { Tells when have a new function to
                                          insure that cursor redisplayed }
        CB : IOCBPtr;                   { Pointer to IOCB used by UnitIO }


Type
        IntTabPtr = ^IntVecTable;
        IntVecTable = array [0..MaxUnit-FakeUnits] of
                        record                  { NO Fake Units Included Here! }
                            SSN: integer;
                            GPtr: integer;
                            Rtn: 0..255;
                            SLink: integer
                        end;

Var
        IntTab: IntTabPtr;              { pointer to interrupt vector table }
        IOPriv1Unused: boolean;         { ***** Unused ***** }
        IOPriv2Unused: boolean;         { ***** Unused ***** }
        KeyEnable: boolean;             { if keyboard interrupts are enabled }


{$ifc Ether3MBaud then}
Var     etherCB: IOCBPtr;               { Pointer to IOCB used by Ethernet }
        pEBuff: IOBufPtr;               { Pointer to Ethernet IO buffer }

        E3Restart,                      { Ether3 state }
        E3IsPromiscuous,
        E3InProgress,
        E3IsReceiving: boolean;
        E3RecErrs: integer;
{$endc}


  { interrupt routines: }

Procedure DiskIntr;     { hard disk }

Procedure FloppyIntr;   { floppy disk }

Procedure SpeechIntr;   { speech out }

Procedure GPIBOutIntr;  { GPIB out }

Procedure GPIBInIntr;   { GPIB in }
```

```
Procedure TabIntr;      { tablet (actually video retrace }

Procedure Z80Intr;      { Z80 monitor }

Procedure KeyIntr;      { keyboard }

Procedure RSIIntr;      { RS232 in }

Procedure RSOIntr;      { RS232 out }

Procedure PutIntr;      { PutStatus completion }

Procedure GetIntr;      { GetStatus completion }

{$ifc Ether3MBaud then}
Procedure Ether3Intr;   { 3 MBaud EtherNet completion }
Function E3Reset: integer;
Procedure E3RecStart;
{$endc}
```

```
Procedure DiskIntr;
```

Abstract: DiskIntr handles a hard disk interrupt by clearing IOInProgress.

```
Procedure FloppyIntr;
```

Abstract: FloppyIntr handles a floppy interrupt by clearing IOInProgress.

```
Procedure SpeechIntr;
```

Abstract: SpeechIntr handles a speech interrupt by clearing IOInProgress.

```
Procedure GPIBOutIntr;
```

Abstract: GPIBOutIntr handles a GPIB output interrupt by clearing IOInProgress.

```
Procedure GPIBInIntr;
```

Abstract: GPIBInIntr handles a GPIB input interrupt. The assumption is that the only GPIB input device is a BitPad. When a GPIB input interrupt is recognized for the first time, the Stanley tablet is turned off. GPIBInIntr gathers characters from the BitPad and updates the tablet buffer.

```
Procedure TabIntr;
```

Abstract: TabIntr (misnamed) handles the video retrace interrupt. It smoothes the tablet data and updates the displayed cursor position (if it is visible and has moved).

```
Procedure Z80Intr;
```

Abstract: Z80Intr ignores Z80 temperature/voltage monitoring.

```
Procedure KeyIntr;
```

Abstract: KeyIntr processes KeyBoard interrupts by copying characters from the KeyBoard buffer into the (misnamed) translated keyboard buffer (KTBuf). Control-C, Control-Shift-C, Control-Shift-D, Control-S, HELP and Control-Q are processed also.

```
function E3Reset: integer;
```

Abstract: E3Reset resets the 3 MBaud EtherNet (?).

```
procedure E3RecStart;
```

Abstract: E3RecStart starts a receive from the 3 MBaud EtherNet (?).

```
Procedure Ether3Intr;
```

Abstract: E3Reset processes a 3 MBaud EtherNet interrupt (?).

```
Procedure RSIIntr;
```

Abstract: RSIIntr ignores an RS232 input interrupt.

```
Procedure RSOIntr;

    Abstract: RSIIntr ignores an RS232 output interrupt.


Procedure PutIntr;

    Abstract: PutIntr handles a PutStatus interrupt by clearing
        IOInProgress.


Procedure GetIntr;

    Abstract: GetIntr handles a GetStatus interrupt by clearing
        IOInProgress.
```

```
dule IO_Unit;


) Unit - Unit IO routines.
les A.  Barel  ca.  1 Jan 80.
pyright (C) 1980, Three Rivers Computer Corporation

stract:
    IO_Unit  exports procedures to perform IO on the various IO Units
    (devices).

    Design: 1) UnitIO must increment and decrement the IOCount of the
    segments  which  are  involved  in  IO.   2)  Segment faults must
    *never* happen while interrupts are off.

rsion Number V6.2
**********}   Exports   {**********}


ports SystemDefs from SystemDefs;


nst
        { Device Code Assignments }
        MaxUnit = 18;           { highest legal device code }
        FakeUnits = 2;          { Number of units which don't have StartIO's }
        IOStart = 0;            { Master Z-80 control }
        HardDisk = 1;
ifc Ether3MBaud then}
        Ether3 = 2;
elsec}
ifc Ether10MBaud then}
        Ether10 = 2;
endc}
endc}

        Floppy = 3;
        Speech = 4;
        GPIBIn = 11;
        GPIBOut = 5;
        Z80Monitor = 6;
        Tablet = 7;
        KeyBoard = 8;
        RS232In = 9;
        RS232Out = 10;
        SpPutSts = 12;          { Put/Get Status }
        SpGetSts = 13;
        SpPutCir = 14;          { Put/Get from Circular Buffer }
        SpGetCir = 15;
        { Fake Units Begin Here }
        TransKey = 16;          { Translated Keyboard }
        ScreenOut = 17;         { Screen Display }
        Clock = 18;             { Used only for Put/Get Status }
        LastUnit = Clock;       { for unit validity checking }
```

```
Type
        IOBufPtr = ^IOBuffer;
        IOBuffer = array[0..0] of integer;

        CBufPtr = ^CBufr;
        CBufr = packed array[0..0] of char;     { same as Memory, except for }
                                                { character buffers }
        BigStr = String[255];                   { A big String }

        UnitRng = 0..MaxUnit;

        IOStatPtr = ^IOStatus;
        IOStatus = record
                HardStatus: integer;            { hardware status return }
                SoftStatus: integer;            { device independent status }
                BytesTransfered: integer
            end;

        IOHeadPtr = ^IOHeader;
        IOHeader = record                       { Hard disk header record }
                       SerialNum: Double;       { Serial number of the file }
                       LogBlock: integer;       { The logical block number }
                       Filler: integer;
                       NextAdr: Double;         { Address of next block in the
                       PrevAdr: Double          { Address of previous block }
                   end;

        DevTabPtr = ^DeviceTable;
        DeviceTable = array [UnitRng] of packed record
                CtlPtr: IOBufPtr;               { actually pointer to ChrCntlBl
                                                { but we'll coerce later }
                BlockSize: integer;             { 0 = variable size }
                                                { 1 = character device (uses ci
                                                { >1 = fixed blocksize (= block
                IntrMask: integer;              { interrupt mask bits }
                IntrPriority: integer;          { decoded iterrupt priority (0.
                PSCode: 0..255;                 { Special code for PutStatus }
                GSCode: 0..255;                 { Special code for GetStatus }
                Name: packed array[0..3] of char
            end;

Const           { RS-232 Speeds }
        RS9600 = 1;
        RS4800 = 2;
        RS2400 = 4;
        RS1200 = 8;
        RS600 = 16;
        RS300 = 32;
        RS150 = 64;
        RS110 = 87;

Type
        Z80Readings = packed record         { Z80 Voltage/Temp Monitor Readings
                Ground: integer;
                Volts5: integer;
                Volts12: integer;
```

```
            Minus12: integer;
            VRef: integer;
            Net: integer;
            CRTemp: integer;
            BaseTemp: integer;
            Volts55: integer;
            Volts24: integer
        end;

    Z80Settings = packed record     { Z80 Voltage/Temp Monitor Settings }
            MinGround: integer;
            MaxGround: integer;
            Min5: integer;
            Max5: integer;
            Min12: integer;
            Max12: integer;
            MinMinus12: integer;
            MaxMinus12: integer;
            MinVRef: integer;
            MaxVRef: integer;
            MinNet: integer;
            MaxNet: integer;
            MinCRTemp: integer;
            MaxCRTemp: integer;
            MinBaseTemp: integer;
            MaxBaseTemp: integer;
            Min55: integer;
            Max55: integer;
            Min24: integer;
            Max24: integer
        end;


    DevStatusBlock = packed record
            ByteCnt: integer;       { # of status bytes }
            case UnitRng of
                KeyBoard,
                Tablet,
$ifc Ether3MBaud then}
                Ether3,
$endc}
                Clock: (DevEnable: boolean);

                RS232In,
                RS232Out: (RSRcvEnable: boolean;
                           RSFill: 0..127;
                           RSSpeed: 0..255;
                           RSParity: (NoParity, OddParity, IllegParity,
                                      EvenParity);
                           RSStopBits: (Syncr,Stop1,Stop1x5,Stop2);
                           RSXmitBits: (Send5,Send7,Send6,Send8);
                           RSRcvBits: (Rcv5,Rcv7,Rcv6,Rcv8));
                Z80Monitor: (Z80Enable: boolean;
                             Z80Fill: 0..32767;
                             case boolean of { Get or Put; true = Get }
```

```
                    true: (        { Get Status }
                           GetRead: Z80Readings;
                           GetLimits: Z80Settings
                          );
                    false:(       { Put Status }
                           PutLimits: Z80Settings
                          ));
          Floppy: (case integer of { Get or Put }
                    1: (           { Get Status }
                        FlpUnit: 0..3;
                        FlpHead: 0..1;
                        FlpNotReady: boolean;
                        FlpEquipChk: boolean;
                        FlpSeekEnd: boolean;
                        FlpIntrCode: 0..3;
                       case integer of
                        1 {IORead, IOWrite, IOFormat}:
                         (FlpMissAddr: boolean; { in data or header }
                          FlpNotWritable: boolean;
                          FlpNoData: boolean;
                          FlpFill1: 0..1;
                          FlpOverrun: boolean;
                          FlpDataError: boolean; { in data or header
                          FlpFill2: 0..1;
                          FlpEndCylinder: boolean;
                          FlpDataMissAddr: boolean; { in data }
                          FlpBadCylinder: boolean;
                          FlpFill3: 0..3;
                          FlpWrongCylinder: boolean;
                          FlpDataDataError: boolean; { in data }
                          FlpFill4: 0..3;
                          FlpCylinderByte:   0..255;
                          FlpHeadByte:       0..255;
                          FlpSectorByte:     0..255;
                          FlpSizeSectorByte: 0..255
                         );
                        2 {IOSeek}:
                         (FlpPresentCylinder: 0..255
                         )
                        );
                    2: (           { Put Status }
                        FlpDensity: 0..255; { single = 0,
                                              double = #100 }
                        FlpHeads: 0..255;   { 1 or 2 heads }
                        FlpEnable: boolean
                       );
                    3: (           { Byte access }
                        FlpByte1: 0..255;
                        FlpByte2: 0..255;
                        FlpByte3: 0..255;
                        FlpByte4: 0..255;
                        FlpByte5: 0..255;
                        FlpByte6: 0..255;
                        FlpByte7: 0..255
                       )
                   )
```

```
            end;


/pe
   IOCommands = (IOReset, IORead, IOWrite, IOSeek, IOFormat,
                 IODiagRead, IOWriteFirst, IOIdle, IOWriteEOI, IOConfigure);


ar
       DevTab: DevTabPtr;                { pointer to system device table }
       CtrlSPending: boolean;            { if ^S has halted screen output }
       IOInProgress: boolean;            { false when speech is active }
       IO24MByte: boolean;               { true if the disk is 24 MBytes }


inction IOCRead(Unit: UnitRng; var Ch: char): integer;
                                         { read a character from a }
                                         { character device and return }
                                         { status: IOB no char available }
                                         {         IOC character returned }
inction IOCWrite(Unit: UnitRng; Ch: char): integer;
                                         { write Ch to character device }
                                         { and return status:           }
                                         {       IOB buffer full        }
                                         {       IOC character sent     }


ocedure UnitIO(Unit: UnitRng;              { IO to block structured }
               Bufr: IOBufPtr;             { devices }
               Command: IOCommands;
               ByteCnt: integer;
               LogAdr: double;
               HdPtr: IOHeadPtr;
               StsPtr: IOStatPtr);

ocedure IOWait(var Stats: IOStatus);       { hang until I/O completes }
inction IOBusy(var Stats: IOStatus): boolean;  { true if I/O not complete }
ocedure IOPutStatus(Unit: UnitRng; var StatBlk: DevStatusBlock);
                                         { Set status on device Unit }
ocedure IOGetStatus(Unit: UnitRng; var StatBlk: DevStatusBlock);
                                         { Reads status on device Unit }
ocedure IOBeep;                          { You guessed it, BEEP! }


ifc Ether3MBaud then}
inction Ether3Transmit(Buff: IOBufPtr; WdCnt: integer) : integer;

inction Ether3Receive(Buff: IOBufPtr; var WdCnt: integer; timeout: integer)
                    : integer;

ction Ether3Start(Promiscuous, Restart: boolean) : integer;
  dc}
```

```
Function IOCRead(Unit: UnitRng; var Ch: char): integer;
```

Abstract:

Reads a character from a character device and returns a completion or error code.

Parameters:

Unit - device from which to read the character.

Ch - character to read.

Returns:

A condition code as defined in the module IOErrors.

```
Function IOCWrite( Unit: UnitRng; Ch: char):integer;
```

Abstract:

Writes a character to a character device and returns a completion or error code. Delays if the buffer is full. Returns an error if the condition doesn't clear up.

Parameters:

Unit - device onto which the character will be written.

Ch - character to write.

Returns:

Condition code as defined by the module IOErrors.

```
Procedure IOWait( var Stats: IOStatus );
```

Abstract:

Hangs until an IO operation inititated by UnitIO is complete.

Parameters:

Stats - Status block that was given to UnitIO when the operation was initiated.

```
Function IOBusy( var Stats: IOStatus): boolean;
```

Abstract:

Determines whether or not I/O is complete.

Parameters:

Stats - Status block that was given to UnitIO when the operation was initiated.

Returns:

True if IO is not complete, false if it is.

```
Procedure IOPutStatus( Unit: UnitRng; var StatBlk:DevStatusBlock );
```

Abstract:

Sets device's characteristics. Has no effect if the device has no settable status.

Parameters:

Unit - device whose characteristics are to be set.

StatBlk - block containing characteristics to be set.

```
Procedure IOGetStatus( Unit: UnitRng; var StatBlk:DevStatusBlock );
```

Abstract:

Reads device status. Has no effect if the device has no readable status.

Parameters:

Unit - device whose characteristics are to be read.

StatBlk - block to which device status is to be returned.

```
Procedure UnitIO( Unit: UnitRng; Bufr: IOBufPtr; Command: IOCommands;
    ByteCnt: integer; LogAdr: double; HdPtr: IOHeadPtr; StsPtr:
    IOStatPtr );
```

Abstract:

IO to non-character devices.

Parameters:

Unit - the device.

Bufr - buffer for data transfers, if requested.

Command - operation to be performed on the device.

ByteCnt - number of bytes to be transferred.

LogAdr - logical address for block structured devices.

HdPtr - pointer to the logical header for operations with the hard disk.

StsPtr - resultant status from the operation.

```
function Ether3Start( promiscuous, restart: boolean): integer;
```

Abstract: Restart the 3 MBaud EtherNet (?).

Parameters: if promiscuous is TRUE, Ether3Receive takes packets for any address. if restart is TRUE, the receiver will be kept active at all times.

Returns: Ether address of this machine.

```
function  Ether3Receive(  Buff: IOBufPtr; var WdCnt: integer; timeout:
     integer): integer;
```

Abstract: Receive data from the 3 MBaud EtherNet (?).

Returns: IOEIOC if successful, WdCnt := size of received packet in words IOEBSE if WdCnt > 512 IOEPTL if a packet > WdCnt is received IOETIM if timeout expires (receiver is still active)
< -16000 hardware status, if receive fails 20 times

```
Function Ether3Transmit( Buff: IOBufPtr; WdCnt: integer): integer;
```

Abstract: Transmit data to the 3 MBaud EtherNet (?).

Returns: IOEIOC if successful IOEBSE if WdCnt > 512
< -16000 hard status, if transmit fails 10 times

```
procedure IOBeep;
```

Abstract:

Causes the PERQ to beep.

```
module Lights;


Lights - Perq Lights.
J.  P.  Strait  26 May 81.
Copyright (C) Three Rivers Computer Corporation, 1981

Abstract:
     This  module  defines the screen coordinates and size of the Perq
     "lights".  These are portions of the  screen  that  are  inverted
     during  tedious  operations  such  as  recalibrating the disk and
     scavenging files (in FileAccess).

     Design: The lights must *not* extend below the 128th line of  the
     screen.  The Y + Size must be less than or equal to 256.  It is a
     good idea for the lights to be totally inside of the title  line.
     The  current  lights start at the left leave lots of room for new
     lights to the right of the current one.  There  is  room  for  10
     lights all together

Version Number V1.2
exports

const

     LightUsed        = TRUE; {whether should use the lights at all}

     LightY           = 3;
     LightHeight      = 14;
     LightWidth       = 18;
     LightSpacing     = 3*LightWidth;

     LightRecalibrate = LightSpacing;
     LightScavenge    = LightRecalibrate + LightWidth + LightSpacing;
     LightSwap        = LightScavenge + LightWidth + LightSpacing;
     LightCompiler    = LightSwap + LightWidth + LightSpacing;
```

```
module Loader;


Loader - Perq system loader.
J.  P.  Strait    10 Feb 81.  rewritten as a module.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     This module implements the  Perq  POS  system  loader.  Given  a
     run-file name as input, it loads and executes that program.  When
     the program terminates (normally or abnormally) it returns to the
     loader which returns to its caller.

Version Number V2.7
exports


   const LoaderVersion = '2.7';


procedure Load( RunFileName: String );
```

```
procedure Load( RunFileName: String );
```

Abstract: Given a run-file name as input, this procedure loads and executes that program. When the program terminates (normally or abnormally) it returns to the loader which returns to its caller.

Parameters:

RunFileName - Name of the .RUN file to load. ".RUN" is appended if it is not already present.

```
module Memory;


Memory - Perq memory manager.
J. P. Strait          1 Jan 80.
Copyright (C) Three Rivers Computer Corporation, 1980.

Abstract:
     Memory is the Perq memory manager.  It supervises the segment
     tables and exports procedures for manipulating memory segments.
     Perq physical memory is segmented into separately addressable
     items (called segments) which may contain either code or data.

     Design: See the Q-Code reference manual.

Version Number V2.13
exports


const MemoryVersion = '2.13';


imports SystemDefs from SystemDefs;
imports Code from Code;


const SATSeg = 1;                   { SAT segment }
      SITSeg = 2;                   { SIT segement }
      FontSeg = 3;                  { font segment }
      ScreenSeg = 4;                { screen segment }
      CursorSeg = 5;                { cursor segment }
      IOSeg = 6;                    { IO segment }
      SysNameSeg = 7;               { system segment names }

      BootedMemoryInBlocks = #1000;     { memory in blocks at boot time }
      MaxSegment = #137;            { should be 2**16 - 1 }

      SetStkBase = #60;
      SetStkLimit = #120;

{$ifc Ether3MBaud then}
      IOSegSize = 6;                { number of blocks in the IOSeg }
{$elsec}
{$ifc Ether10MBaud then}
      IOSegSize = 3;                { number of blocks in the IOSeg }
{$elsec}
      IOSegSize = 3;                { number of blocks in the IOSeg }
{$endc}
{$endc}

      SysSegLength = 8;            { length of name of a boot-loaded segment }

      MMMaxBlocks = #400;           { maximum number of blocks in a segment
      MMMaxCount = #377;
      MMMaxIntSize = MMMaxBlocks-1;
      MMMaxExtSize = MMMaxBlocks;
```

```
type MMBit4 = 0..#17;
     MMBit8 = 0..#377;
     MMBit12 = 0..#7777;
     MMIntSize = 0..MMMaxIntSize;
     MMExtSize = 1..MMMaxExtSize;
     MMAddress = integer;
     MMPosition = (MMLowPos, MMHighPos);

     SegmentNumber = integer;

     SegmentKind = (CodeSegment, DataSegment);

     SegmentMobility = (UnMovable, UnSwappable, LessSwappable, Swappable);

     MMFreeNode = record
      N: MMAddress;
      L: integer
      end;

     MMBlockArray = array[0..0] of array[0..127] of integer;

     pMMBlockArray = ^MMBlockArray;

     MMArray = record case Integer of
                1: (m: array[0..0] of MMFreeNode);
                2: (w: array[0..0] of Integer)
                end;

     pMMArray = ^MMArray;

     MMPointer = record case integer of
      1: (P: ^integer);
      2: (B: pMMBlockArray);
      3: (M: pMMArray);
      4: (Offset: MMAddress;
          Segmen: SegmentNumber)
      end;

     SATentry = packed record { Segment Address Table }
      { **** ENTRIES MUST BE TWO WORDS LONG **** }
      NotResident : boolean;              { 001 }
      Moving      : boolean;              { 002 }
      RecentlyUsed: boolean;              { 004 }
      Sharable    : boolean;              { 010 }
      Kind        : SegmentKind;          { 020 }
      Full        : boolean;              { 040 }
      InUse       : boolean;              { 100 }
      Lost        : boolean;  { *** }     { 200 }
      BaseLower   : MMBit8;
      BaseUpper   : MMBit4;
      Size        : MMBit12
      end;
```

```
    SITentry = packed record case integer of { Segment Information Table
     { **** ENTRIES MUST BE EIGHT WORDS LONG **** }
     1: { real SIT entry }
        (NextSeg     : SegmentNumber;
         RefCount    : 0..MMMaxCount;
         IOCount     : 0..MMMaxCount;
         Mobility    : SegmentMobility;
         BootLoaded  : Boolean;
         SwapInfo    : record case {BootLoaded:} Boolean of
                         True:  (BootLowerAddress: Integer;
                                 BootUpperAddress: Integer;
                                 BootLogBlock: Integer);
                         False: (DiskLowerAddress: Integer;
                                 DiskUpperAddress: Integer;
                                 DiskId: Integer)
                         end;
         case SegmentKind of
           DataSegment: (Increment  : MMIntSize;
                         Maximum    : MMIntSize;
                         Freelist   : MMAddress);
           CodeSegment: (Update     : TimeStamp)
        );
     2: { boot time information }
        (BootBlock: record
           CS: SegmentNumber;      { initial code segment }
           SS: SegmentNumber;      { initial stack segment }
           XX: Integer;            { unused }
           VN: Integer;            { system version number }
           FF: SegmentNumber;      { first free segment number }
           FC: SegmentNumber;      { first system code segment }
           DK: integer;            { disk system was booted from }
           CH: integer             { char used in booting }
           end)
     end;

    SATarray = array[0..0] of SATentry;

    SITarray = array[0..0] of SITentry;

    pSAT = ^SATarray;

    pSIT = ^SITarray;

    MMEdge = record
             H: SegmentNumber;  { Head }
             T: SegmentNumber   { Tail }
             end;

    SysSegName = packed array[1..SysSegLength] of Char;

    pSysNames = ^SysNameArray;

    SysNameArray = array[0..0] of SysSegName;


procedure InitMemory;
```

```
procedure DataSeg( var S: SegmentNumber );
procedure CodeOrDataSeg( var S: SegmentNumber );
procedure ChangeSize( S: SegmentNumber; Fsize: MMExtSize );
procedure CreateSegment( var S: SegmentNumber;
                         Fsize, Fincrement, Fmaximum: MMExtSize );
procedure IncRefCount( S: SegmentNumber );
procedure SetMobility( S: SegmentNumber; M: SegmentMobility );
procedure DecRefCount( S: SegmentNumber );
procedure SetIncrement( S: SegmentNumber; V: MMExtSize );
procedure SetMaximum( S: SegmentNumber; V: MMExtSize );
procedure SetSharable( S: SegmentNumber; V: boolean );
procedure SetKind( S: SegmentNumber; V: SegmentKind );
procedure MarkMemory;
procedure CleanUpMemory;
procedure FindCodeSegment( var S: SegmentNumber; Hint: SegHint );
procedure EnableSwapping( Where: Integer );
procedure DisableSwapping;
function  CurrentSegment: SegmentNumber;

exception UnusedSegment( S: SegmentNumber );
```

Abstract:

UnusedSegment is raised when the memory manager encounters a segment number which references a segment which is not in use. This may mean that a bad segment number was passed to some memory manager routine or that a bad address was de-referenced.

Parameters:

S - Segment number of the unused segment.

```
exception NotDataSegment( S: SegmentNumber );
```

Abstract:

NotDataSegment is raised when the number of a code segment is passed to some memory manager routine that requires the number of a data segment.

Parameters:

S - Segment number of the code segment.

```
exception BadSize( S: SegmentNumber; Fsize: Integer );
```

Abstract:
BadSize is raised when a bad Size value is passed to some memory manager routine. This usually means that the size passed to CreateSegment or ChangeSize is greater than the maximum size or less than one.

Parameters:
Fsize - The bad Size value.

```
exception BadIncrement( S: SegmentNumber; Fincrement: Integer );
```

Abstract:
BadIncrement is raised when a bad Increment value is passed to some memory manager routine. This usually means that the increment passed to CreateSegment is greater than 256 or less than one.

Parameters:
Fincrement - The bad Increment value.

```
exception BadMaximum( S: SegmentNumber; Fmaximum: Integer );
```

Abstract:
BadMaximum is raised when a bad Maximum value is passed to some memory manager routine. This usually means that the maximum passed to CreateSegment is greater than 256 or less than one.

Parameters:
Fmaximum - The bad Maximum value.

```
exception FullMemory;
```

Abstract:
FullMemory is raised when there is not enough physical memory to satisfy some memory manager request. This is raised only after swapping segments out and compacting memory.

```
exception CantMoveSegment( S: SegmentNumber );
```

Abstract:
CantMoveSegment is raised when the memory manager attempts to move a segment which is UnMovable or has a non-zero IO count.

Parameters:
S - The number of the segment which cannot be moved.

```
exception PartNotMounted;
```

Abstract:
PartNotMounted is raised when 1) the memory manager attempts to swap a data segment out for the first time and 2) the partition which is to be used for swapping is no longer mounted.

```
exception SwapInFailure( S: SegmentNumber );
```

Abstract:
SwapInFailure is raised when the swap file cannot be found for a segment which is marked as swapped out. This is an error which should never happen in a debugged system. It usually means that there is a bug in the memory manager or that the segment tables have been clobbered.

Parameters:
S - The number of the segment which could not be swapped in.

```
exception EdgeFailure;
```

Abstract:
EdgeFailure is raised by MakeEdge when it discovers that the SIT entries are not linked together into a circular list. This is an error which should never happen in a debugged system. It usually means that there is a bug in the memory manager or that the segment tables have been clobbered.

```
exception NilPointer;
```

Abstract:
NilPointer is raised when a Nil pointer is used or passed to Dispose.

```
exception BadPointer;
```

Abstract:
BadPointer is raised when a bad pointer is passed to Dispose.

Parameters:

```
exception FullSegment;
```

Abstract:
FullSegment is raised by New when it discovers that there is not enough room to allocate and the segment cannot be enlarged (its size has reached its maximum).

```
exception NoFreeSegments;
```

Abstract:
NoFreeSegments is raised when the memory manager discovers that all of the segment numbers are in use and it needs another one. This is equivalent to "Segment table full".

```
exception SwapError;
```

Abstract:
SwapError is raised if the one of the memory managers swapping routines is called when swapping is disabled. This is an error which should never happen in a debugged system. It usually means that there is a bug in the memory manager.

```
var SAT: pSAT;
    SIT: pSIT;
    MMFirst, MMFree, MMLast, MMHeap: SegmentNumber;
```

```
    MMHole: MMEdge;
    MMState: (MMScan1, MMScan2, MMScan3, MMScan4, MMScan5,
              MMScan6, MMScan7, MMScan8, MMScan9, MMScan10,
              MMScan11,
              MMNotFound, MMFound);
    StackSegment: SegmentNumber;
    FirstSystemSeg: SegmentNumber;
    BootFileId: Integer;
    SwappingAllowed: Boolean;
    SwapId: Integer;
    MemoryInBlocks: Integer;  { amount of memory on this machine }
```

```
procedure InitMemory;
```

Abstract: InitMemory initializes the memory manager. It is called once at system initialization and may not be called again. If the system was booted from a floppy, the system segments are all marked as UnSwappable.

```
procedure DataSeg( var S: SegmentNumber );
```

Abstract: DataSeg is used to
1) - Determine if a given segment number represents a data segment.
2) - Find the default heap segment (in the case of an input parameter of zero).

Parameters:
S - Data segment number--zero means the default heap segment.

Errors:
UnusedSegment - if S is not in use.
NotDataSegment - if S is not a data segment.

```
procedure CodeOrDataSeg( var S: SegmentNumber );
```

Abstract: CodeOrDataSeg is used to
1) - Determine if a given segment number represents a defined segment
2) - Find the default heap segment (in the case of an input parameter of zero).

Parameters:
S - Data segment number--zero means the default heap segment.

Errors: UnusedSegment if S is not in use.

```
procedure ChangeSize( S: SegmentNumber; Fsize: MMExtSize );
```

Abstract: ChangeSize is used to change the size of an existing data segment.

Parameters:
- S - Number of the segment whose size is to be changed.
- Fsize - New size of the segment.

Errors:
- UnusedSegment - if S is not in use.
- BadSize - if Fsize is greater than the maximum size of S or less than one.
- FullMemory - if there is not enough physical memory to increase the size of S.
- CantMoveSegment - if the segment must be moved, but it is not movable or its IOCount is not zero.

```
procedure CreateSegment( var S: SegmentNumber; Fsize, Fincrement,
    Fmaximum: MMExtSize );
```

Abstract: CreateSegment is used to create a new data segment.

Parameters:
- S - Set to the number of the new segment.
- Fsize - Desired size of the new segment in blocks.
- Fincrement - Increment size of the new segment in blocks.
- Fmaximum - Maximum size of the new segment.

Errors:
- BadSize - if Fsize is greater than Fmaximum or less than one.
- BadIncrement - if Fincrement is greater than MMMaxExtSize or less than one.
- BadMaximum - if Fmaximum is greater than MMMaxExtSize or less than one.
- FullMemory - if there is not enough physical memory to create the segment.

```
procedure IncRefCount( S: SegmentNumber );
```

Abstract: IncRefCount increments the number of references to a segment. A
non-zero reference count prevents a segment from being destroyed. A reference count greater than one indicates a system segment.

Parameters:
S - Number of the segment.

Errors: UnusedSegment if S is not in use.

```
procedure SetMobility( S: SegmentNumber; M: SegmentMobility );
```

Abstract: SetMobility sets the Mobility of a segment. The mobility may be set to one of the following values:
Swappable - segment is a candidate for swapping or moving.
LessSwappable - segment is a candidate for swapping or moving, but the memory manager will be more reluctant to swap.
UnSwappable - segment may not be swapped, but may be moved.
UnMovable - segment may not be swapped or moved. The RecentlyUsed bit of the segment is cleared also. Thus to make a segment a candidate for swapping, set its mobility to Swappable (even if it already swappable).

Parameters:
S - Segment number.
M - Mobility.

Errors:
UnusedSegment - if S is not in use.
CantMoveSegment - if the segment is changing from Swappable to UnMovabl an attempt is made to move the segment to the high end of memory. If it has a non-zero IO count, or swapping is disabled, this error is issued.
FullMemory - if the segment is changing from Swappable to UnSwappable, it is swapped out, and there isn't enough memory to swap it in.

```
procedure DecRefCount( S: SegmentNumber );
```

Abstract: DecRefCount decrements the reference count of a segment by one. If reference and IO counts both become zero:
- if the segment is a data segment, it is destroyed.
- if the segment is a code segment, it is destroyed only if it is in the screen or is non-resident.

Parameters:
S - Number of the segment.

Errors: UnusedSegment if S is not in use.

```
procedure SetIncrement( S: SegmentNumber; V: MMExtSize );
```

Abstract: SetIncrement changes the increment size of a data segment.

Parameters:
S - Number of the segment.
V - New increment size.

Errors:
UnusedSegment - if S is not in use.
NotDataSegment - if S is not a data segment.
BadIncrement - if V is greater than MMMaxExtSize or less than one.

```
procedure SetMaximum( S: SegmentNumber; V: MMExtSize );
```

Abstract: SetMaximum changes the maximum size of a data segment.

Parameters:
S - Number of the segment.
V - New maximum size.

Errors:
UnusedSegment - if S is not in use.
NotDataSegment - if S is not a data segment.
BadMaximum - if V is greater than MMMaxExtSize or less than one.

```
procedure SetSharable( S: SegmentNumber; V: boolean );
```

Abstract: SetSharable changes the "sharable" attribute of a segment (this attribute is not currently used).

Parameters:
S - Number of the segment.
V - New value of the "sharable" attribute.

Errors:
UnusedSegment - if S is not in use.

```
procedure SetKind( S: SegmentNumber; V: SegmentKind );
```

Abstract: SetKind changes the kind (code or data) of a segment.

Parameters:
S - Number of the segment.
V - New kind of the segment.

Errors:
UnusedSegment - if S is not in use.

```
procedure MarkMemory;
```

Abstract: MarkMemory marks all currently in use segments as system segments usually before loading a user program) by incrementing their reference counts.

```
procedure CleanUpMemory;
```

Abstract: CleanUpMemory destroys all user segments (usually at the end of a program execution) by deecrementing the reference count of all segments.

```
procedure EnableSwapping( Where: Integer );
```

Abstract: EnableSwapping turns the swapping system on, determines where swap files should be created, and locates the boot file.

Parameters:
Where - FileId of some file in the partition to be used for swap files.

```
procedure DisableSwapping;
```

Abstract: DisableSwapping attempts to swap in all segments which are swapped out and then turns the swapping system off. If there is not enough physical memory to swap all segments in, swapping is not disabled.

Errors:
FullMemory - if there isn't enough memory to swap all segments in.

```
procedure FindCodeSegment( var S: SegmentNumber; Hint: SegHint );
```

Abstract: FindCodeSegment searches for a code segment in the segment table which has a certain SegHint. If such a segment is found, its RefCount is incremented and the segment number is returned. Otherwise, a zero segment number is returned.

Segments which 1) Have a DiskId equal to Hint.FId, 2) Have an Update date/time not equal to Hint.Update, and 3) Have a RefCount of zero are deleted. This is done because such a segments reference code files which have been overwritten and are no longer valid. Such segments will not get the memory manager into trouble, but they will never be used again, and it is just as well to get rid of them.

**** Hint must be a valid hint. That is, the file specified **** by Hint.FId must have a FileWriteDate equal to Hint.Update.

Parameters:
S - Return parameter set to zero or the number of the code segment.
Hint - Desired SegHint.

```
function CurrentSegment: SegmentNumber;
```

Abstract: CurrentSegment finds the segment number of its caller.

Result: CurrentSegment = Segment number of the caller of CurrentSegment.

```
module MoveMem;


MoveMem - Move memory.
J. P. Strait  ca. 1 Jan 80.
Copyright (C) Three Rivers Computer Corporation, 1980, 1981.

Abstract:
     MoveMem is used to move a segment from one location to another in
     physical memory.  The two locations may overlap.

Version Number V1.8
exports


imports Memory from Memory;


procedure CopySegment
               ( SrcSeg, DstSeg: SegmentNumber; NewDstBase: Integer );
```

```
procedure CopySegment( SrcSeg, DstSeg: SegmentNumber; NewDstBase:
    Integer );
```

Abstract: CopySegment is used to move a segment from one location to another in physical memory.

Parameters:

SrcSeg - Number of the segment which represents the source address and source size.

DstSeg - Number of the segment which represents the destination address.

NewDstBase - New value of the base address for DstSeg.

Result: Base address of SrcSeg - set to the old base address of DstSeg. Base address of DstSeg - set the NewDstBase.

Design: CopySegment moves segments without swapping them, and is designed in such a way that it may move itself or its own stack. In order to be able to do this, CopySegment is in a code segment by itself so that any call is guaranteed to be a cross-segment call. Thus when the move operation is complete, a cross segment return is done, and the CodeBase micro-code register is reloaded. Movemem executes a special StartIO instruction to cause the micro-code to reload its StackBase register. This code segment must never exceed 256 words in length so that when CopySegment moves itself, the new copy cannot overlap the old one.

CopySegment uses RasterOp to copy the memory in order that the copy be done as an indivisible operation.

DO NOT CHANGE this routine unless you fully understand the entire system.

```
Module MultiRead;


Written by Brad Myers    24 Jul 81.

Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     This Module exports a procedure to read a file very quickly  into
     memory.  The memory for the blocks of the file to be read must be
     allocated contiguously before the call is made.  Typically,  this
     will be done by using CreateSegment.

Version Number V1.0
{//////////////////////////} EXPORTS {\\\\\\\\\\\\\\\\\\\\\\\\\\}

Imports FileSystem from FileSystem;

Procedure MultiRead
             (fid: FileID; addr: pDirBlk; firstBlock,numBlocks: integer);
```

```
Procedure MultiRead(fid: FileID; addr: pDirBlk;  firstBlock,numBlocks:
     integer);
```

Abstract: Does a multi-sector read on the file specified into the memory pointed to by addr NOTE: This only works for contiguous files.

Parameters: fid - the fileID of the file to read from.
addr - the address of the start of the memory to read the file into. This must be pre-allocated.
firstBlock - the logical block number of the first to read (the first legal value is 0; -1 will not work).
numBlocks - the count of the number of blocks to transfer.

```
module PasLong;


PasLong - Extra stream package input conversion routines.
J. P. Strait & Michael R. Kristofic ca. 15 Sep 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     PasLong is the extra character input module of the Stream
     package. Its routines are called by code generated by the Pascal
     compiler in response to variations on Read, Readln, Write and
     Writeln statements. It is one level above Module Stream and uses
     Stream's lower-level input routines.

Version Number V2.2
exports


imports Stream from Stream;


procedure ReadD( var F: FileType; var X: long; B: integer );
procedure WriteD( var F: FileType; X : long; Field, B: integer );
```

```
procedure ReadD( var F: FileType; var X: long; B: integer );
```

Abstract: Reads an double integer in free format with base B. B may be any integer between 2 and 36, inclusive.

Parameters:
- X - the double to be read.
- F - the file from which X is to be read.
- B - the base of X. It may be any integer between 2 and 36, inclusive. If B is less than zero and the user does not type an explicit plus or minus sign, X is read as an unsigned number.

Errors:
- PastEof -if an attempt is made to read F past the Eof.
- NotNumber - if non-numeric input is encountered in the file.
- LargeNumber - if the number is not in the range -2^31..2^32-1.
- BadBase - if the base is not in 2..36.

Design: Number is read into the low order word of two double precision integers to avoid overflow.

```
procedure WriteD( var F: FileType; X: long; Field, B: integer );
```

Abstract: Writes an double integer in fixed format with base B.

Parameters:
- X - the double to be written.
- F - the file into which X is to be written.
- Field - the size of the field into which X is to be written.
- B - the base of X. It is an integer whose absolute value must be between 2 and 36, inclusive. If B is less than zero, X is written as an unsigned number.

Errors:
- BadBase -if the base is not in 2..36.

Design: Value written from two double precision words to avoid overflow.

```
module PasReal;


PasReal -  Scott L.  Brown      Created: 25-Nov-81
Copyright (C) 1981 - Three Rivers Computer Corporation

Abstract:

     PasReal is an extra character input module of the Stream package.
     Its routines are called by code generated by the Pascal  compiler
     in  response  to  variations  on  Read, Readln, Write and Writeln
     statements.  It  is  one  level  above  Module  Stream  and  uses
     Stream's lower-level input routines.

Version Number V0.1
exports

 imports Stream from Stream;


procedure ReadR(var F: FileType;
                var value: real);

procedure WriteR(var F: FileType;
                     e: real;
                     TotalWidth: integer;
                     FracDigits: integer;
                     format: integer);
```

```
procedure ReadR(var F: FileType; var value: real);
```

Abstract:

This procedure reads a real number from the file F, and returns its value.

Parameters:

F - identifies the file from which to read.

value - is a return parameter returning the value of the real number read from the file F.

Results:

This procedure modifies the file buffer for F, and stores the value of the real number in value.

Side Effects:

This procedure reads characters from the external file F, until it receives a character which cannot be part of the real number, and it leaves this character in the file buffer.

There has been only minimal care taken with the file buffer, so if an exception is raised during the read, there is no guarantee about its contents.

Exceptions:

PastEof - raised if an attempt is made to read beyond the end of file.

NotReal - raised if the stream of characters in file F does not correspond to a real number.

SmallReal - raised if the number read is too small to be represented by a 32-bit IEEE-Standard real number.

LargeReal - raised if the number read is too large to be represented by a 32-bit IEEE-Standard real number.

Calls:

StreamName - in module Stream for the stream name of file F.

GetC - in module Stream for reading the next character from

file F.

RealMul

TenPower - returns a real number representing 10.0 raised to an integer (range -37..38) power.

Design:

The regular expression for real numbers is described by a DFA and implemented by a case statement, where each element of the case statement corresponds to a state in the DFA. This case statement stores information about the number read into (primarily) three variables: 1) mant - the mantissa of the number read (up to 2**24-1), 2) scale_factor - any adjustment to the mantissa (as a power of ten), and 3) exp - the exponent of the number read.

Effort is taken to ignore zeros whenever possible. If they are insignificant they are ignored, if significant they cause an adjustment to the scale_factor and do not affect the mantissa. The variables save_scale_factor, save_mant, zero_count, frac_part and sig_digit_in_frac are used for this purpose.

To combine this information into a real number, the scale_factor is added to the exponent and this sum is the power of ten exponent of the mantissa. To combine sum with the mantissa, it has to be converted to a real number, which is done by the function TenPower. Then the result of TenPower is multiplied with the mantissa to produce the real number returned as value.

Much care has been taken to avoid calls to TenPower with actual parameters which are out of range, and also to minimize the number of real multiplications necessary (this to avoid error propogation).

```
procedure WriteR(var F: FileType; e: real; TotalWidth: integer;
    FracDigits: integer; format: integer);
```

Abstract:

This procedure writes a real number to a file F, under the given format specifications.

Parameters:

F - the file to which to write.

e - the value to write.

TotalWidth - the minimum number of characters to write.

FracDigits - the number of characters in the fractional part (used for fixed format only).

format - indicates either fixed or floating format.

Results:

The file buffer for F is modified by procedures nested in this one.

Calls:

StreamName - in module Stream for the stream name of file F.

base2_to_base10

normalize

Design:

There is some initialization, then if the real number to be written is zero, eWritten and ExpValue are obviously zero, else the real number needs to be converted to base 10 giving eWritten and ExpValue. Then a call is made to a procedure for the desired format.

The design here follows as much as possible the ISO Standard description for writing Pascal real numbers, in particular, the choice of many variable names.

```
Module PERQ_String;


PERQ String manipulation routines.
Written by: Donald Scelza
Copyright (C) 1980
Three Rivers Computer Corporastion

Abstract:
     This module implements the string manipulation routines  for  the
     Three Rivers PERQ Pascal.

     Strings in PERQPascal are stored a single character per byte with
     the byte indexed by 0 being the length of the string.   When  the
     routines  in  this  module must access the length byte, they must
     turn off range checking.

Version Number V2.4
{*********************}  Exports  {*********************}


Const MaxPStringSize=255;           { Length of strings}
Type PString = String[MaxPStringSize];

Procedure Adjust(Var STR:PString;  LEN:Integer);
Function Concat(Str1,Str2:PString):PString;
Function Substr(Source:PString;  Index,Size: Integer):PString;
Procedure Delete(Var Str:PString;  Index,Size:Integer);
Procedure Insert(Var Source,Dest:PString; Index:Integer);
Function Pos(Source,Mask:PString): Integer;

FUNCTION PosC(s: PString; c: Char): Integer;
PROCEDURE AppendString(var s1: PString; s2: PString);
PROCEDURE AppendChar(var s: PString; c: Char);
FUNCTION UpperCase(c: Char): Char;
PROCEDURE ConvUpper(Var s: PString);

Exception StrBadParm;

     Abstract:  Raised  when  bad index or length parameters passed to
          procedures or
               sometimes when string will be too  long  (other  times,
               StrLong is raised in this case

Function RevPosC(s: PString; c: char): integer;
```

Procedure Adjust(var Str:PString; Len:Integer);

Abstract: This procedure is used to change the dynamic length of a string.

Parameters: Str is the string that is to have the length changed.

Len is the new length of the string. This parameter must be This value must be no greater than MaxPStringSize.

Environment: None

Results: This procedure does not return a value.

Side Effects: This procedure will change the dynamic length of Str.

Errors: If Len > MaxPStringSize then raise StrLong exception.

Design: Simple.

Function Concat(Str1,Str2:PString):PString;

Abstract: This procedure is used to concatenate two string together.

Parameters: Str1 and Str2 are the two strings that are to be concatenated.

Environment: None

Results: This function will return a single string as described by the parameters.

Errors: If Length(Str1) + Length(Str2) is greater then MaxPStringSize then raise StrLong exception.

Design:

Function SubStr(Source:PString; Index, Size:Integer):PString;

Abstract: This procedure is used to return a sub portion of the string passed as a parameter.

Parameters: Source is the string that we are to take a portion of.

Index is the starting position in Source of the substring.

Size is the size of the substring that we are to take.

Environment: None

Results: This function returns a substring as described by the parameter list.

Errors: If Index or Size are greater than MaxPStringSize then raise StrBadParm exception.

Design:

Procedure Delete(var Str:PString; Index, Size:Integer);

Abstract: This procedure is used to remove characters from a string.

Parameters: Str is the string that is to be changed. Characters will be removed from this string.

Index is the starting position for the delete.

Size is the number of character that are to be removed. Size characters will be removed from Str starting at Index.

Environment: None

Results: This procedure does not return a value.

Side Effects: This procedure will change Str.

Errors: None

Design:

Procedure Insert(var Source, Dest:PString; Index:Integer);

Abstract: This procedure is used to insert a string into the middle of another string.

Parameters: Source is the string that is to be inserted.

Dest is the string into which the inseration is to be made.

Index is the starting position, in Dest, for the inseration.

Environment: None

Results: This procedure does not return a value.

Side Effects: This procedure will insert Source in Dest starting at location Index.

Errors: If the resulting string is too long then generate a runtime error.

Design:

PROCEDURE AppendString(var s1: PString; s2: PString);

Abstract: puts s2 on the end of s1

Parameters : s1 is the left String and s2 goes on the end.

Calls: PerqString.Concat.

SideEffects : modifies s1.

```
PROCEDURE AppendChar(var s: PString; c: Char);
```

Abstract: puts c on the end of s

Parameters : s is the left String and c goes on the end.

SideEffects : modifies s.

```
FUNCTION UpperCase(c: Char): Char;
```

Abstract: Changes c to uppercase if letter.

Parameters: c is any char.

Returns: char is uppercase if letter otherwise unchanged.

```
Procedure ConvUpper(Var s: PString);
```

Abstract: Converts s to all upper case

Parameters: s, passed by reference, to be converted

```
Function PosC(s: PString; c: char): integer;
```

Abstract: Tests if c is a member of s.

Parameters: c is any char; s is string to test for c member of.

Returns: index of first c in s (from beginning of string) or zero if not there.

```
Function RevPosC(s: PString; c: char): integer;
```

Abstract: Tests if c is a member of s.

Parameters: c is any char; s is string to test for c member of.

Returns: index of first c in s (from end of string) or zero if not there.

```
Function Pos(Source, Mask:PString):Integer;
```

Abstract: This procedure is used to find the position of a pattern in a given string.

Parameters: Source is the string that is to be searched.

Mask is the pattern that we are looking for.

Environment: None

Results: If Mask occured in Source then the index into Source of the first character of Mask will be returned. If Mask was not found then return 0.

Side Effects: None

Errors: None

Design:

```
module PMatch;


Abstract:
     Does pattern matching on strings

     Patterns accepted are as follows:


     "*" matches 0 or more characters.

     "&" matches 1 or more characters.

     "#" matches exactly 1 character.

     "'0" matches any digit.

     "'A" matches any alphabetic (capitals only unless casefold).

     "'a" matches any alphabetic(lower case only unless 'casefold').

     "'@" matches any non-alphanumeric.

     "'*" matches '*', other patterns chars can be quoted also.

     Written by: Gene Ball at CMU
Version Number V2.5
{////////////////////////////} Exports {\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\}

Type pms255 = String[255];

Function PattMatch(var str,pattern: pms255; fold: boolean): boolean;
Function PattMap(var str,inpatt,outpatt,outstr:pms255; fold:boolean): boolean;
Procedure PattDebug(v: boolean);
Function IsPattern(var str: pms255): boolean;

Exception BadPatterns;

     Abstract: Raised if outPatt and  inPatt  do  not  have  the  same
          patterns
               in the same order for PattMap
```

```
Procedure PattDebug(v: boolean);
```

Abstract: Sets the global debug flag

Parameters: v is value to set debug to

SideEffects: Changes debug value

```
Function IsPattern(var str: pms255): boolean;
```

Abstract: Tests to see whether str contains any pattern matching characters

Parameters: str - string to test. If not pattern then removes all quotes.

Returns: true if str contains any pattern matching characters; else false

```
Function PattMatch(var str, pattern: pms255; fold: boolean): boolean;
```

Abstract: Compares str against pattern

Parameters: str - full string to compare against pattern;
pattern - pattern to compare against. It can have special characters in it
fold - determines whether upper and lower case are distinct. If true then not.

Returns: true string matches pattern; false otherwise

```
Function PattMap(var str,inpatt,outpatt,
    outstr:pms255;fold:boolean):boolean;
```

Abstract: Compares str against inPatt, putting the parts of str that match inpatt into the corresponding places in outpatt and returning the result EXAMPLES:

PattMap('test9.pas', 'test'0.pas', 'xtest'0.pas') => TRUE, 'xtest9.pas'

PattMap('test9.pas', '*.pas', '*.ada') => TRUE, 'test9.ada'

Parameters: str - full string to compare against pattern;
inpatt - pattern to compare against. It can have special characters in it
outpatt - pattern to put the parts of str into; it must have the same special characters in the same order as in inpatt
outStr - the resulting string if PattMap returns true;

fold-determines whether upper and lower case are distinct. It true then not.

Returns: true string matches pattern; false otherwise

Errors: Raises BadPatterns if outPatt and inPatt do not have the same patterns in the same order

```
Module PopCmdParse;


Abstract:
     This module provides procedures to help with PopUp menus.  See
     the module PopUp for the definition of  pNameDesc  and  for  some
     useful procedures for creating and destroying pNameDescs.

     Written by Brad Myers    Nov 18, 1981

     Copyright (C) 1981 Three Rivers Computer Corperation

Version Number V1.8
{*********************} Exports {**********************}

Imports CmdParse from CmdParse;
Imports PopUp from PopUp;

Function PopUniqueCmdIndex(Cmd: CString; Var names: pNameDesc): Integer;

Function GetCmdLine(Procedure IdleProc; prompt: String;
                    var line, cmd: CString; var inF: pCmdList;
                    var names: pNameDesc; var firstPress: boolean;
                    popOK: boolean): integer;

Function GetShellCmdLine(var cmd: CString; var inF: pCmdList;
                  var names: pNameDesc): integer;

Function GetConfirm(Procedure IdleProc; popOK: boolean;
                     prompt: String; def: integer;
                     var switches: pSwitchRec): integer;

Procedure NullIdleProc;
```

Procedure NullIdleProc;

Abstract: This procedure does nothing. It is useful as an IdleProc parameter to other procedures when no IdleProc is needed.

Function PopUniqueCmdIndex(Cmd: CString; Var names: pNameDesc): Integer;

Abstract: This procedure is used to do a unique lookup in a popUp command table. It is the same as UniqueCmdIndex except the table of names is the kind used by popUp menus. If cmd is the full name of one of the names in names, even if is also a sub-part of other names, it is returned as the one found.

Parameters: Cmd - the command that we are looking for.

CmdTable - a table of the valid commands. The first valid command in this table must start at index 1.

NumCmds - the number of valid command in the table.

Returns: The index of Cmd in CmdTable. If Cmd was not found then return NumCmds + 1. If Cmd was not unique then return NumCmds+2.

Function GetShellCmdLine(var cmd: CString; var inF: pCmdList; var names: pNameDesc): integer;

Abstract: This routine is similar to GetCmdLine except that it works on the command line specified to the Shell. It is should be used by programs that use GetCmdLine to parse the Shell command line. Command files are handled by GetShellCmdLine. The user can call ParseCmdArgs after GetShellCmdLine to get the arguments to the command.

Parameters: cmd - the first command taken off the line. This will be valid even if the return value is greater than numCommands. It will be '' if no command found before the first significant break character.

inF - a command file list created by InitCmdFile. Just call InitCmdFile and pass in the inF returned. This procedure manages the list and handles all command files.

names - a variable length array of names used for popUp menus and for matching the input cmd against.

Returns: Identical to GetCmdLine. Viz: index in the array or

numCommands + 1 ==> Name not found in array

numCommands + 2 ==> Name not unique

numCommands + 3 ==> Name was empty

numCommands + 4 ==> First command was a switch (it is in Cmd).

numCommands + 5 ==> Illegal character found after command.

```
Module PopUp;


Written by  Brad A.  Myers 16-Nov-80

Abstract: This program produces pop up windows that replace the screen
     area  at a specified cursor location.  The cursor is then changed
     and PopUp waits for a press.  Whenever the cursor is  inside  the
     window,  the command at that point is highlighted.  If a press is
     done inside the window, the highlighted command is selected.  The
     user  can  control whether one or more than one command should be
     selected before window is removed.  If a press  outside  outside,
     no  command  is  executed.  In any case, the window is erased and
     the original contents of that area is returned

     Copyright (C) 1981 - The Three Rivers Computer Corporation
Version Number V2.4
{-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-} EXPORTS {-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-}

EXCEPTION BadMenu;

     Abstract: Raised when parameters are illegal.

EXCEPTION Outside;

     Abstract: raised when press outside of menu.

Type s25 = String[25];
     NameAr = Array[1..1] of s25;
     pNameAr = ^NameAr;
     NameDesc = Record
                   header: s25;
                   numCommands: integer;
                   commands: NameAr;
                End;
     pNameDesc = ^NameDesc;

     ResRes = ^ResArray;
     ResArray = Record
                   numIndices: integer;
                   indices: Array[1..1] of integer;
                End;

PROCEDURE Menu(names: pNameDesc; isList: boolean;
               first, last, curX, curY, maxYsize: integer; VAR res: ResRes

PROCEDURE InitPopUp;
PROCEDURE DestroyRes(var res: ResRes);

PROCEDURE AllocNameDesc(numNames, seg: Integer; Var names: pNameDesc);
PROCEDURE DestroyNameDesc(Var names: pNameDesc);
```

```
Procedure AllocNameDesc(numNames, seg: Integer; Var names: pNameDesc);
```

Abstract: There are two ways to allocate the storage for a NameDesc. One is to declare in your program a type with an array of the correct size and the other fields exactly the same way. You then RECAST a pointer to that array into a pNameDesc. The other way is to use this procedure. It allocates the storage for numNames out of a segment. Turn off range checking when assigning or accessing the array.

NOTE: To deallocate the nameDesc returned, use DestroyNameDesc

Parameters: numNames - the number of names in the array.

seg - the segment to allocate the nameDesc out of. If zero, then uses the default segment. This procedure uses NewP so the segment can have other things in it also.

names - set with the newly allocated pNameDesc. Its numCommands field is set with numNames. Do not change this size or the deallocation will not work.

```
Procedure DestroyNameDesc(Var names: pNameDesc);
```

Abstract: Delete names. It should have been created by AllocateNameDesc. The numCommands field better be the same as set when allocated.

Parameters: names - The storage for names is deallocated and names is set to NIL.

```
PROCEDURE InitPopUp;
```

Abstract: creates cursors needed to make PopUp windows work. This should be called once before calling menu.

Environment: sets cursors.

```
PROCEDURE DestroyRes(var res: ResRes);
```

Abstract: Deallocates storage for res and sets it to NIL

Parameters: res is ResRes to destroy

PROCEDURE Menu(names: pNameDesc; isList: boolean; first, last, curX, curY, maxYsize: integer; VAR res: ResRes);

Abstract: puts up a window with commands commands stacked vertically with the center at curX, curY. Allocates off the heap enough storage for old picture at that place so can restore it. Deallocates all storage when done.

Parameters: names - a pointer to an array of names to put in the menu. In it

header - is put at the top of the menu. It may be empty in which case there is no header.

numcommands - the number of names in the array.

commands - an array of names to display. These can be generated by having pNameDesc with an array of the correct (or larger than the correct) size and recasting it into a pNameDesc, or by creating a segment to hold all the names that will be needed.

isList - if true says that a number of commands can be selected. if false, Menu returns as soon as the first command is selected.

first - the index of the first command in names^.commands to display. To display all items, use 1.

last - the index of the last command in names^.commands to display. To display all items, use names^.numCommands. Last must be greater than first.

curX - the x position at which to display the menu. If -1 then uses current pen position.

curY - the y position at which to display the menu. If -1 then uses current pen position.

maxYsize - the maximum size in bits of the menu. If -1, then menu will be big enough to hold all items (up to the size of the screen). maxYsize must be greater than 4*(fontHeight+4) which is 68 for the default font.

res - is set with an answer array. This array is allocated off the heap by Menu. Use DestroyRes to deallocate it. If Menu is exited via ^C or a press outside, then res is not allocated. The fields of res are set as follows:

numIndices - the number of items selected. If not isList then will be 1. If isList, will not be zero

indices - a variable length array of the indices of the names chosen. They are in increasing order irrespective of the order the names were picked.

Errors: Catches CtlC and raises CtlCAbort after removing the menu. Catches CtlShiftC and erases menu then re-raises CtlShiftC. Catches HelpKey and erases menu and then re-raises. If continued, then raises OutSide. Raises OutSide if press outside of the menu window. Raises BadMenu if parameters are illegal.

Environment: Requires enough memory be on the heap for picture. Requires that InitPopup has been called.

```
module Profile;


Abstract:
     This  module  is  used  to  get information from the user profile
     file.

     Written by: Don Scelza

     Copyright (C) Three Rivers Computer Corperation,  1981

Version Number V1.1
{********************} Exports {********************}


     This module provides facilities that will allow a program to get informati
     from the user profile.

     The profile file is a text file that has the form:

     #<Subsystem name> <Line of text for that sub system>  <More  text  for  th
     subsystem> - - - #<Next subsystem>---

     The base unit of the file is a text line.  The function that provides valu
     from the profile file will return a line  of  text  each  time  that  it
     called.    All  text  line  between  the  #<Subsystem  name>  and  the  ne
     #<Subsystem name> are assumed to be assoicated  with  the  first  subsys
     Successive  calles  to  PFileEntry will return the next line of text for
     current subsystem.


Exception PNotFound(FileName: String);

     Abstract: Raised when profile file cannot be found

     Parameters: fileName is profile not found

Exception PNotInited;

     Abstract: Raised when a profile procedure is used  but  PFileInit
          not called
               first

Type ProfStr = String[255];

procedure PFileInit(PFileName, SubSystem: ProfStr);

function PFileEntry: ProfStr;
```

```
procedure PFileInit(PFileName, SubSystem: ProfStr);
```

Abstract: This procedure is called each time a subsystem wishes to start to read information from the profile file. It is only called once per subsystem invocation. It will lookup the profile file and search for the required subsystem.

Parameters: PFileName is the name of the profile file that is to be used.

SubSystem is the name of the subsystem that is to be searched for.

Side Effects: This procedure will change Inited, InLine and PFile.

Errors: If PFileName was not found then raise PNotFound.

```
function PFileEntry: ProfStr;
```

Abstract: This procedure is used to get the next profile entry for a subsystem.

Results: This procedure will return the next line from the profile file for the current subsystem. If there are no more lines for the current subsystem return null.

Environment: PFileInit must have been called before this procedure is used. Uses the global InLine. Sets InLine to be empty.

Errors: If PFileInit was not called then raise PNotInited.

```
module QuickSort;
```


Written by: Mark G. Faust

Abstract:

Hoare's Quicksort algorithm with some simple optimizations. This module provides two procedures, IntegerSort and StringSort, which sort arrays of integers and strings respectively.

For a detailed description of the algorithm and references to papers on its analysis see [Robert Sedgewick, "Implementing Quicksort Programs," in CACM 21(10), 1978.]

Because of rigid type checking of arrays in Pascal, pointers to the arrays to be sorted are passed along with an integer specifying the length of the array. The procedures require that the array be declared [0..N+1] where the 0th through Nth elements are to be sorted. The additional array element is used to speed up the sorting routine. Before passing the array pointer to the sort procedure it is RECAST as either a IntegerArrayPtr or a StringArrayPtr. An example for the integer sort is given below. The string sort is analogous.

```
program ShowSort(input,output);

imports QuickSort from QuickSort;

const Size = 99;

type MyArray = array[0..Size+1] of integer;

var MyArrayPtr :^MyArray; i :integer;

begin new(MyArrayPtr);

for i := 0 to Size do readln(MyArrayPtr^[i]);

IntegerSort(Size,recast(MyArrayPtr,PIntArray));

for i := 0 to Size do writeln(MyArrayPtr^[i]); end.
```

```
exports

type

    ss25 = String[25];
    IntArray = array[0..0] of integer;
    StrArray = array[0..0] of ss25;

    PIntArray = ^IntArray;
    PStrArray = ^StrArray;
```

```
procedure IntegerSort(N :integer; A :PIntArray);
procedure StringSort(N :integer; A: PStrArray; Fold :boolean);
```

```
procedure IntegerSort(N :integer; A :PIntArray);
```

Abstract: Given an integer N and a pointer to an array [0..N+1] of integers, sort [0..N] into ascending order using QuickSort.

Parameters: N :integer One less than the upper bound of the array. It is the largest index of a valid key.

A :PIntArray A pointer to an array [0..N+1] of integers.

Results: The array from [0..N] is in ascending order

Side Effects: The array is sorted and the N+1st element contains MaxInt

```
procedure StringSort(N :integer; A :PStrArray; Fold :boolean);
```

Abstract: Given an integer N and a pointer to an array [0..N+1] of strings, sort [0..N] into ascending lexigocraphic order using QuickSort. StringSort is case sensitive (e.g. A < a) unless Fold is True.

Parameters: N :integer One less than the upper bound of the array. It is the largest index of a valid key.

A :PStrArray A pointer to an array [0..N+1] of strings.

Fold :boolean If True then we fold to UpCase for comparisons.

Results:

The array from [0..N] is in ascending order

Side Effects:

The array is sorted and the N+1st element contains the DEL character

```
module RandomNumbers;


J. P. Strait    15 Sep 80.
Copyright (C) Three Rivers Computer Corporation, 1980.


Module RandomNumbers contains two routines:

InitRandom - initializes the random number generator.

Random - a function which returns a new random number each time
it is referenced.

There is currently no way to seed the generator.

Random is a feedback shift-register pseudo-random number generator.
The algorithm used is one described in the article:

'Generalized Feedback Shift Register Pseudorandom Number
Generator'
T. G. Lewis and W. H. Payne
JACM Vol. 20, No. 3, July 1973, pp. 456-468.

Random produces multidimensional pseudo-random numbers equally
distributed in the interval -32768..32767 and has a period of
2^98.
Version Number V1.2
{////////////////////////////} exports {\\\\\\\\\\\\\\\\\\\\\\\\\\}

procedure InitRandom;
function Random: integer;
```

```
Procedure InitRandom;
```

Abstract: Initialize the random number generator. Every time this is called, the random numbers start over at the same place.

```
MODULE Raster;


Copywrite (C) 1980 - The Three Rivers Computer Corporation
EXPORTS

  Const RRpl     = 0;        { Raster Op function codes }
        RNot     = 1;
        RAnd     = 2;
        RAndNot  = 3;
        ROr      = 4;
        ROrNot   = 5;
        RXor     = 6;
        RXNor    = 7;

  Type RasterPtr = ^RasterArray; {a pointer that can be used as RasterOp
                                  or Line source and destination }
       RasterArray = Array[0..0] of integer;
```

```
Module ReadDisk;


Abstract:

     Module to Read and write to the disk using a buffer system

     Written by the CMU Spice Group
Version Number V1.5
{******************} exports {****************************}
imports DiskIO from DiskIO;

function  ReadDisk(addr : DiskAddr)     : ptrDiskBuffer;
function  ChangeDisk(addr : DiskAddr)   : ptrDiskBuffer;
function  ReadHeader(addr : DiskAddr)   : ptrHeader;
function  ChangeHeader(addr : DiskAddr) : ptrHeader;
procedure FlushDisk(addr : DiskAddr);
procedure WriteDisk(addr : DiskAddr; ptr : ptrDiskBuffer; hdptr : ptrHeader);
procedure WriteHeader(addr : DiskAddr; ptr : ptrDiskBuffer; hdptr : ptrHeader);

procedure InitBuffers;
function  FindDiskBuffer(dskaddr : DiskAddr; alwaysfind : boolean) : integer;
procedure ReleaseBuffer(indx : integer);
procedure FlushBuffer(indx : integer);
procedure FlushAll;
procedure ChangeBuffer(indx : integer);
procedure ChgHdr(indx : integer);
procedure UseBuffer(indx,numtimes : integer);
function  BufferPointer(indx : integer) : ptrDiskBuffer;
function  HeaderPointer(indx : integer) : ptrHeader;
function  ReadAhead(addr : DiskAddr)    : ptrDiskBuffer;
procedure ForgetAll;

Exception FlushFail(msg: String; operation: DiskCommand; addr: DiskAddr;
                    softStat: integer);

     Abstract: Raised when the system is unable to flush out a buffer.
          The buffer
               is  marked  as  flushed out, however, so the error will
               not repeat the next time a buffer needs to be flushed

     Parameters: Same as DiskFailure (in DiskIO)

     Resume:  ALLOWED,  but  has  no  effect  (procedure  will  return
          normally as if
               flush had been successful)
```

Function ReadDisk(addr : DiskAddr) : ptrDiskBuffer;

Abstract: Read the block specified and return the ptr of the buffer read into

Parameters: addr is the address of the block to read

Returns: ptr to buffer read into

Function ReadAhead(addr : DiskAddr) : ptrDiskBuffer;

Abstract: Identical to ReadDisk

Parameters: addr is the address of the block to read

Returns: ptr to buffer read into

Function ReadHeader(addr : DiskAddr) : ptrHeader;

Abstract: Reads block specified and returns a ptr to a buffer describing its header

Parameters: addr is the address of the block to read

Returns: ptr to header read into

Function ChangeDisk(addr : DiskAddr) : ptrDiskBuffer;

Abstract: Reads block specified and returns a ptr to its data; in addition, mark file as changed so flush will write it out

Parameters: addr is the address of the block to read

Returns: ptr to buffer holding the block read into

Function ChangeHeader(addr : DiskAddr) : ptrHeader;

Abstract: Reads block specified and returns a ptr to its data; in addition, mark file as header changed so flush will write it out using IOWriteFirst

Parameters: addr is the address of the block to read

Returns: ptr to header read into

```
Procedure FlushDisk(addr : DiskAddr);
```

Abstract: Removes block specified from buffer system and writes it out if changed. If addr not in buffer then NO-OP

Parameters: addr is block to flush

```
Procedure WriteDisk(addr : DiskAddr; ptr : ptrDiskBuffer; hdptr :
    ptrHeader);
```

Abstract: Writes out a block using DskWrite. If block for addr is in a buffer then Release it first.

Parameters: addr - the address of the block to write
ptr - points to a buffer of data
hdptr - points to a buffer of header

```
Procedure WriteHeader(addr : DiskAddr; ptr : ptrDiskBuffer; hdptr :
    ptrHeader);
```

Abstract: Writes out a block using DskFirstWrite. If block for addr is in a buffer then Release it first.

Parameters: addr is the address of the block to write ptr points to a buffer of data hdptr points to a buffer of header

```
Procedure InitBuffers;
```

Abstract: Initializes the buffer system

```
function FindDiskBuffer(dskaddr : DiskAddr; alwaysfind : boolean) :
    integer;
```

Abstract: Finds the buffer that contains the data for block dskAddr.

Parameters: dskAddr - is address to find buffer for alwaysFind tells whether to read in if not found

Returns: Index of buffer found or zero if not there

```
Procedure ReleaseBuffer(indx : integer);
```

Abstract: Mark the table entry as unused.

Parameters: indx - is entry to mark

```
Procedure FlushBuffer(indx : integer );
```

Abstract: Write out the data for the buffer indx if changed and then mark the buffer as not changed.

Parameters: indx - is buffer to flush

Errors: FlushFail - is raised if cannot Flush a buffer due to a write error

```
Procedure FlushAll;
```

Abstract: Writes out the data for all the buffers and then mark them all as unchanged.

Errors: FlushFail - is raised if cannot Flush a buffer due to a write error. Does not stop at first error, but goes and tries all buffers before raising the exception

```
Procedure ChgHdr(indx : integer);
```

Abstract: Mark a buffer as having its header changed.

Parameters: indx - is buffer to mark

```
Procedure UseBuffer(indx,numtimes : integer);
```

Abstract: Mark a buffer as used.

Parameters: indx - is buffer to mark
numTimes - the number to incrememt use count by

```
Function BufferPointer(indx : integer) : ptrDiskBuffer;
```

Abstract: return the bufferPtr for a buffer.

Parameters: indx - is buffer

Returns: Ptr to buffer

```
Function HeaderPointer(indx : integer) : ptrHeader;
```

Abstract: return the header Ptr for a buffer.

Parameters: indx - is buffer

Returns: ptr to header

```
module Reader;


Reader - Stream package input conversion routines.
J.  P.  Strait   ca.  1 Jan 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     Reader is the character input module of the Stream  package.   It
     is called by code generated by the Pascal compiler in response to
     Read or Readln.  It is one level above  Module  Stream  and  uses
     Stream's lower-level input routines.

Version Number V2.1
exports


imports Stream from Stream;


procedure ReadBoolean( var F: FileType; var X: boolean );
procedure ReadCh( var F: FileType; var X: char; Field: integer );
procedure ReadChArray( var F: FileType; var X: ChArray; Max, Len: integer
procedure ReadIdentifier( var F: FileType; var X: integer;
                          var IT: IdentTable; L: integer );
procedure ReadInteger( var F: FileType; var X: integer );
procedure ReadString( var F: FileType; var X: String; Max, Len: integer
procedure ReadX( var F: FileType; var X: integer; B: integer );
```

```
procedure ReadBoolean( var F: FileType; var X: boolean );
```

Abstract: Reads a boolean in free format.

Parameters:
X - the boolean to be read.
F - the file from which X is read.

Errors: PastEof if an attempt is made to read F past the Eof.
NotBoolean if a non-boolean is encountered in the file.

```
procedure ReadCh( var F: FileType; var X: char; Field: integer );
```

Abstract: Reads a character in fixed or free format.

Parameters:
X - the character to be read.
F - the file from which X is to be read.
Field - the size of the field X is in.

Errors: PastEof if an attempt is made to read F past the Eof.

```
procedure ReadChArray( var F: FileType; var X: ChArray; Max, Len:
    integer );
```

Abstract: Reads a packed character array in free or fixed format. If free format reading is selected, spaces are skipped and characters are read until another space is encountered.

Parameters:
X - the character array to be read.
F - the file from which X is to be read.
Max - the declared length of X.
Len - the size of the field. Len <= 0 selects free format reading.

Errors: PastEof if an attempt is made to read F past the Eof.

```
procedure ReadIdentifier( var F: FileType; var  X:  integer;  var  IT:
     IdentTable; L: integer );
```

Abstract: Reads an identifier and returns its position in a table. A table lookup is performed requiring only that the identifier typed uniquely matches the beginning of a single table entry.

Parameters:
- X - set to the ordinal of the identifier read.
- F - the file from which X is read.
- IT - the table of identifiers indexed from 0 to L.
- L - the largest identifier ordinal defined by the table.

Errors:
- BadIdTable - if length of the identifier table is less than 1.
- PastEof - if an attempt is made to read F past the Eof.
- NotIdentifier - if a non-identifier is encountered in the file.
- IdNotDefined - if the identifier is not in the table.
- IdNotUnique - if the identifier is not unique.

```
procedure ReadInteger( var F: FileType; var X: integer );
```

Abstract: Reads a decimal integer in free format.

Parameters:
- X - the integer to be read.
- F - the file from which X is to be read.

Errors:
- PastEof - if an attempt is made to read F past the Eof.
- NotNumber - if non-numeric input is encountered in the file.
- LargeNumber- if the number is not in the range -32768..32767.

```
procedure ReadString( var  F:  FileType;  var  X:  String;  Max,  Len:
     integer );
```

Abstract: Reads a string in free or fixed format. If free format is selected, spaces are skipped and characters are read until another space is encountered.

Parameters:
- X - the string to be read.
- F - the file from which X is to be read.
- Max - the declared maximum length of the string.
- Len - the size of the field. Len <= 0 selects free format.

Errors:
- PastEof - if an attempt is made to read F past the Eof.

```
procedure ReadX( var F: FileType; var X: integer; B: integer );
```

Abstract: Reads an integer in free format with base B. B may be any integer between 2 and 36, inclusive.

Parameters:
- X - the integer to be read.
- F - the file from which X is to be read.
- Field - the size of the field X is in.
- B - the base of X. It may be any integer between 2 and 36, inclusive.

Errors:
- PastEof - if an attempt is made to read F past the Eof.
- NotNumber- if non-numeric input is encountered in the file.
- LargeNumber- if the number is not in the range -32768..32767.
- BadBase - if the base is not in 2..36.

```
module RealFunctions;


RealFunctions - Standard functions for reals.
J. Strait  27 Nov 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     RealFunctions implements many of the standard functions whose
     domain and/or range is the set of real numbers.  The
     implementation of these functions was guided by the book

     Software Manual for the Elementary Functions, William J. Cody,
     Jr. and William Waite, (C) 1980 by Prentice-Hall, Inc.

     The domain (inputs) and range (outputs) of the functions are
     given in their abstract.  The following notation is used.
     Parentheses () are used for open intervals (those that do not
     include the endpoints), and brackets [] are used for closed
     intervals (those that do include their endpoints).  The closed
     interval [RealMLargest, RealPLargest] is used to mean all real
     numbers, and the closed interval [-32768, 32767] is used to mean
     all integer numbers.

     Currently all functions described by Cody and Waite are
     implemented with the exception of the hyperbolic functions (SinH,
     CosH, TanH).

     DISCLAIMER:

     Only the most cursory testing of these functions has been done.
     No guarantees are made as to the accuracy or correctness of the
     functions.  Validation of the functions must be done, but at some
     later date.

     Design: AdX, IntXp, SetXp, and Reduce are implemented as Pascal
     functions.  It is clear that replacing the calls with in-line
     code (perhaps through a macro expansion) would improve the
     efficiency.

     Many temporary variables are used.  Elimination of unnecessary
     temporaries would also improve the efficiency.

     Many limit constants have been chosen conservatively, thus
     trading a small loss in range for a guarantee of correctness.
     The choice of these limits should be re-evaluated by someone with
     a better understanding of the issues.

     Some constants are expressed in decimal (thus losing the
     guarantee of precision).  Others are expressed as Sign, Exponent,
     and Significand and are formed at execution time.  Converting
     these two 32-bit constants which are Recast into real numbers
     would improve the correctness and efficiency.

     More thought needs to be given to the values which are returned
     after resuming from an exception.  The values that are returned
```

now are the ones recommended by Cody and Waite. It seems that Indefinite values (NaNs in the IEEE terminology) might make more sense in some cases.

Version Number V1.0
exports

```
const RealPInfinity =   Recast(#17740000000,Real);   {  1.0 / 0.0 }
      RealMInfinity =   Recast(#37740000000,Real);   { -1.0 / 0.0 }
      RealPIndefinite = Recast(#00000000001,Real);   {  0.0 / 0.0 }
      RealMIndefinite = Recast(#20000000001,Real);   { -0.0 / 0.0 }
      RealPLargest =    Recast(#17737777777,Real);   { largest positive }
      RealMLargest =    Recast(#37737777777,Real);   { largest negative }
      RealPSmallest =   Recast(#00040000000,Real);   { smallest positive }
      RealMSmallest =   Recast(#20040000000,Real);   { smallest negative }

function Sqrt( X: Real ): Real;
function Exp( X: Real ): Real;
function Ln( X: Real ): Real;
function Log10( X: Real ): Real;
function Power( X, Y: Real ): Real;
function PowerI( X: Real; Y: Integer ): Real;
function Sin( X: Real ): Real;
function Cos( X: Real ): Real;
function Tan( X: Real ): Real;
function CoTan( X: Real ): Real;
function ArcSin( X: Real ): Real;
function ArcCos( X: Real ): Real;
function ArcTan( X: Real ): Real;
function ArcTan2( Y, X: Real ): Real;



exception SqrtNeg( X: Real );
```

Abstract:

SqrtNeg is raised when Sqrt is passed a negative argument. You may resume from this exception, in which case Sqrt returns Sqrt(Abs(X)).

Parameters:

X - Argument of Sqrt.

```
exception ExpLarge( X: Real );
```

Abstract:
ExpLarge is raised when Exp is passed an argument which is too large. You may resume from this exception, in which case Exp returns RealPInfinity.

Parameters:
X - Argument of Exp.

```
exception ExpSmall( X: Real );
```

Abstract:
ExpLarge is raised when Exp is passed an argument which is too small. You may resume from this exception, in which case Exp returns 0.0.

Parameters:
X - Argument of Exp.

```
exception LogSmall( X: Real );
```

Abstract:
LogSmall is raise when Ln or Log10 is passed an argument which is too small. You may resume from this exception in which case Ln or Log10 returns RealMInfinity if X is zero or the log of Abs(X) if X is non-zero.

Parameters:
X - Argument of Ln or Log10.

```
exception PowerZero( X, Y: Real );
```

Abstract:
PowerZero is raised when Power or PowerI is called with X = 0.0 and Y = 0.0. You may resume from this exception in which case Power or PowerI returns RealPInfinity.

Parameters:
X - Argument of Power or PowerI. Y - Argument of Power or PowerI.

exception PowerNeg(X, Y: Real);

Abstract:
PowerNeg is raised when Power is called with X < 0.0 or with X = 0.0 and Y < 0.0, or PowerI is called with X = 0.0 and Y < 0. You may resume from this exception in which case Power or PowerI returns Power(Abs(X),Y) in the case of X < 0.0 or returns RealPInfinity in the case of X = 0.0 and Y < 0.0.

Parameters:
X - Argument of Power or PowerI. Y - Argument of Power or PowerI.

exception PowerBig(X, Y: Real);

Abstract:
PowerBig is raised when Power or PowerI is called with X and Y for which X raised to the Y power is too large to be represented. You may resume from this exception in which case Power or PowerI returns RealPInfinity.

Parameters:
X - Argument of Power or PowerI. Y - Argument of Power or PowerI.

exception PowerSmall(X, Y: Real);

Abstract:
PowerSmall is raised when Power or PowerI is called with X and Y for which X raised to the Y is too close to zero to be represented. You may resume from this exception in which case Power or PowerI returns 0.0.

Parameters:
X - Argument of Power or PowerI. Y - Argument of Power or PowerI.

```
exception SinLarge( X: Real );
```

Abstract:
SinLarge is raised when Sin is called with an argument which is too large. You may resume from this exception in which case Sin returns 0.0.

Parameters:
X - Argument of Sin.

```
exception CosLarge( X: Real );
```

Abstract:
CosLarge is raised when Cos is called with an argument which is too large. You may resume from this exception in which case Cos returns 0.0.

Parameters:
X - Argument of Cos.

```
exception TanLarge( X: Real );
```

Abstract:
CosLarge is raised when Tan or CoTan is called with an argument which is too large. You may resume from this exception in which case Tan or CoTan returns 0.0.

Parameters:
X - Argument of Tan or CoTan.

```
exception ArcSinLarge( X: Real );
```

Abstract:
ArcSinLarge is raised when ArcSin is called with an argument which is too large. You may resume from this exception in which case ArcSin returns RealPInfinity.

Parameters:
X - Argument of ArcSin.

```
exception ArcCosLarge( X: Real );
```

Abstract:
ArcCosLarge is raised when ArcCos is called with an argument which is too large. You may resume from this exception in which case ArcCos returns RealPInfinity.

Parameters:
X - Argument of ArcCos.

```
exception ArcTan2Zero( Y, X: Real );
```

Abstract:
ArcTan2Zero is raised when ArcTan2 is called with both X and Y equal to zero. You may resume from this exception in which case ArcTan2 returns RealPInfinity.

Parameters:
Y - Argument of ArcTan2. X - Argument of ArcTan2.

```
function Sqrt( X: Real ): Real;
```

Abstract: Compute the square-root of a number.

Domain = [0.0, RealPLargest]. Range = [0.0, Sqrt(RealPLargest)].

Parameters:
X - Input value.

Returns:
Square-root of X.

```
function Ln( X: Real ): Real;
```

Abstract: Compute the natural log of a number.

Domain = [0.0, RealPLargest]. Range = [RealMLargest, Ln(RealPLargest)].

Parameters:
X - Input value.

Returns: Natural log of X.

```
function Log10( X: Real ): Real;
```

Abstract: Compute the log to the base 10 of a number.

Domain = [0.0, RealPLargest]. Range = [RealMLargest, Log10(RealPLargest)].

Parameters:
X - Input value.

Returns: Log to the base 10 of X.

```
function Exp( X: Real ): Real;
```

Abstract: Compute the exponential function.

Domain = [-85.0, 87.0]. Range = (0.0, RealPLargest].

Parameters:
X - Input value.

Returns: e raised to the X power.

```
function Power( X, Y: Real ): Real;
```

Abstract: Compute the result of an arbitrary number raised to an arbitrary power.

DomainX = [0.0, RealPLargest]. DomainY = [RealMLargest,RealPLargest]. Range = [0.0, RealPLargest].

With the restrictions that 1) if X is zero, Y must be greater than zero. 2) X raised to the Y is a representable real number.

Parameters:
X - Input value.
Y - Input value.

Returns: X raised to the Y power.

```
function PowerI( X: Real; Y: Integer ): Real;
```

Abstract: Compute the result of an arbitrary number raised to an arbitrary integer power. The difference between Power and PowerI is that negative values of X may be passed to PowerI.

DomainX = [RealMLargest, RealPLargest]. DomainY = [-32768, 32767]. Range = [RealMLargest, RealPLargest].

With the restrictions that 1) if X is zero, Y must be non-zero. 2) X raised to the Y is a representable real number.

Parameters:
X - Input value.
Y - Input value.

Returns: X raised to the Y power.

```
function Sin( X: Real ): Real;
```

Abstract: Compute the sin of a number.

Domain = [-1E5, 1E5]. Range = [-1.0, 1.0].

Parameters:
X - Input value.

Returns: Sin of X.

```
function Cos( X: Real ): Real;
```

Abstract: Compute the cosin of a number.

Domain = [-1E5, 1E5]. Range = [-1.0, 1.0].

Parameters:
X - Input value.

Returns: Cos of X.

```
function Tan( X: Real ): Real;
```

Abstract: Compute the tangent of a number.

Domain = [-6433.0, 6433.0]. Range = [RealMInfinity, RealPInfinity].

Parameters:
X - Input value.

Returns: Tangent of X.

```
function CoTan( X: Real ): Real;
```

Abstract: Compute the cotangent of a number.

Domain = [-6433.0, 6433.0]. Range = [RealMInfinity, RealPInfinity].

Parameters:
X - Input value.

Returns: Cotangent of X.

```
function ArcSin( X: Real ): Real;
```

Abstract: Compute the arcsin of a number.

Domain = [-1.0, 1.0). Range = [-Pi/2, Pi/2).

Parameters:
X - Input value.

Returns: Arcsin of X.

Design: It seems that the Domain and Range ought to be closed intervals, however this implementation apparently returns a number very close to zero when X is 1.0, rather than returning Pi/2 as it should.

```
function ArcCos( X: Real ): Real;
```

Abstract: Compute the arccosin of a number.

Domain = (-1.0, 1.0]. Range = (-Pi/2, Pi/2].

Parameters:
X - Input value.

Returns: Arccosin of X.

Design: It seems that the Domain and Range ought to be closed intervals, however this implementation apparently returns a number very close to zero when X is -1.0, rather than returning -Pi/2 as it should.

```
function ArcTan( X: Real ): Real;
```

Abstract: Compute the arctangent of a number.

Domain = [RealMLargest, RealPLargest]. Range = (-Pi/2, Pi/2).

Parameters:
X - Input value.

Returns: Arctangent of X.

Design: Seems fine except for very large numbers.

```
function ArcTan2( Y, X: Real ): Real;
```

Abstract: Compute the arctangent of the quotient of two numbers. One interpretation is that the parameters represent the cartesian coordinate (X,Y) and ArcTan2(Y,X) is the angle formed by (X,Y), (0,0), and (1,0).

DomainY = [RealMLargest, RealPLargest]. DomainX = [RealMLargest, RealPLargest]. Range = [-Pi, Pi].

Parameters:
Y - Input value.
X - Input value.

Returns: Arctangent of Y / X.

Design: Seems fine except for very large Y/X.

```
module RS232Baud;

Abstract:

     RS232Baud - set RS232 baud rate with optional input  enable.   J.
     P.   Strait       21 Aug 80.  Copyright (c) Three Rivers Computer
     Corporation 1980.
Version Number V1.1
exports

procedure SetBaud(Baud: String; Enable: Boolean);

Exception BadBaudRate;

     Abstract: Raised if string is not a valid baud rate
```

```
procedure SetBaud(Baud: String; Enable: Boolean);

     Abstract: Sets the baud rate to baud specified by string arg
          Arguments: Baud is string of new baud rate (e.g. "2400")
          Enable says whether to allow transfers on RS232

     SideEffects: Changes status of RS232

     Errors: Raises BadBaudRate if string is illegal
```

```
module RunRead;


RunRead - Module to read run files.
John P Strait   9 Apr 81.
CopyRight (C) Three Rivers Computer Corporation, 1981.

Abstract:
     RunRead exports procedures to read and write run files.

     Design: If and when the format of run files is changed, the
     constant RFileFormat in module Code must be changed. This is
     necessary so that the procedures to read run files will not fail.

Version Number V1.1
exports


const RunReadVersion = '1.1';


imports Code from Code;


procedure ReadRunFile( var RunFile: RunFileType; Seg: Integer;
                       var Header: RunInfo;
                       var FirstSeg, FirstUserSeg, LastSeg: pSegNode;
                       ImportsWanted: Boolean );

procedure ReadSegNames( var RunFile: RunFileType; Seg: Integer;
                        FirstUserSeg: pSegNode );
```

```
procedure ReadRunFile( var RunFile: RunFileType; Seg: Integer; var
    Header: RunInfo; var FirstSeg, FirstUserSeg, LastSeg: pSegNode;
    ImportsWanted: Boolean );
```

Abstract: ReadRunFile reads a run file and builds a structure that represents that run file. The run file is read up to, but not including, the names of the .Seg files.

Parameters:

- RunFile - A file variable which has been Reset to the desired file. ReadRunFile does *not* close the file.
- Seg - Segment number for dynamic allocation.
- Header - The RunInfo record.
- FirstSeg - Set to point to the first segment in the run file.
- FirstUserSeg - Set to point to the first user segment in the run file.
- LastSeg - Set to point to the last segment in the run file.
- ImportsWanted - True iff Import entries are to be read from the run file.

```
procedure ReadSegNames( var RunFile: RunFileType; Seg: Integer;
    FirstUserSeg: pSegNode );
```

Abstract: ReadSegNames reads .Seg file names from a run file and adds them to a structure that represents that run file.

Parameters:

- RunFile - A file variable which has been Reset to the desired file and already read with ReadRunFile. ReadSegNames does *not* close the file.
- Seg - Segment number for dynamic allocation.
- FirstUserSeg - A pointer to the first user segment in the run file.

```
module RunWrite;


RunWrite - Module to write run files.
John P Strait   9 Apr 81.
CopyRight (C) Three Rivers Computer Corporation, 1981.

Abstract:
     RunWrite exports procedures to write run files.

     Design: If and when the format of run files is changed, the
     constant RFileFormat in module Code must be changed.  This is
     necessary so that the procedures to read run files will not crap
     out.

Version Number V1.1
exports


const RunWriteVersion = '1.1';


imports Code from Code;



procedure WriteRunFile( var RunFile: RunFileType; Header: RunInfo;
                        FirstSeg, FirstUserSeg: pSegNode );
```

```
procedure  WriteRunFile(  var  RunFile:  RunFileType; Header: RunInfo;
     FirstSeg, FirstUserSeg: pSegNode );
```

Abstract: ReadRunFile writes a run file from a structure that represents that run file.

Parameters:

- RunFile - A file variable which has been Rewritten to the desired file. WriteRunFile does *not* close the file.
- Header - The RunInfo record.
- FirstSeg - A pointer to the first segment in the run file.
- FirstUserSeg - A pointer to the first user segment in the run file.

```
Module Screen;
Written By: Miles A.  Barel    July 1, 1980
Three Rivers Computer Corporation
Pittsburgh, PA 15213
Abstract: Provides the interface to the PERQ screen including
     rudimentary support for multiple windows

Exports
Imports Raster from Raster;
Version Number V3.12
Const   ScreenVersion = 'V3.12';
        VarWin = false;
                 {if true then can have an arbitrary number of windows
                  and storage for them has to be allocated off a heap.
                  If false then there are 17 windows max, and
                  storage is in screens global data.
                  NOTE: There are still bugs in VarWin true}
Type    FontPtr = ^Font;
        Font = Packed Record    { Contains character sets }
                Height:integer; { Height of the KSet }
                Base: integer;
                          { distance from top of characters to base-line }
                Index: Array [0..#177] of { Index into character patterns }
                    Packed Record case boolean of
                      true: (Offset: 0..767;
                          { position of character in patterns
                        Line: 0..63;
                          { Line of patterns containing char }
                        Width: integer); { Width of the character }
                       false:(Loc:integer; Widd: integer)
                    end;
                Filler: array[0..1] of integer;
                Pat: Array [0..0] of integer;  { patterns go here }
                                  { We turn off range checking to }
                                  { access patterns, hence allowing }
                                  { KSets of different sizes }
            end;
{$ifc VarWin then}
      WindowP = ^WindowType;
{$endc}
      WindowType = Packed Record
{$ifc VarWin then}
                     winNumber: Integer;  {this window number}
{$endc}
                     winBY, winTY, winLX, winRX, { Limits of window area }
                     winHX, winHY, winMX, winMY, { Limits of useable area }
                     winCurX, winCurY, winFunc: integer;
                     winKSet: FontPtr;
                     winCrsChr: char;
                     winHasTitle, winCursorOn, defined: boolean;
{$ifc VarWin then}
                     winNext: WindowP;
{$endc}
                     end;
{$ifc VarWin then}
```

```
Const   MaxWIndx = 32767;
{$elsec}
Const   MaxWIndx = 17;
{$endc}
Type    WinRange = 0..MaxWIndx;
        LineStyle = (DrawLine,EraseLine,XorLine);
        LS = String[255];
Procedure ScreenInit;           { CALL THIS ONCE AT BOOT }
Procedure ScreenReset;
                    { This procedure de-allocates storage for
                     all windows and sets up the default window.  }
Procedure SPutChr(CH:char);    { put character CH out to current position }
                               { on the screen.  Chars FF, CR, and LF }
                               { have special meanings unless #200 bit set:
                               {       FF - clear screen
                               {       CR - move left to margine
                               {       LF - move vertically down one
                               {       BS - erase previous character   }
Procedure SSetCursor(X,Y: integer);     { Set Cursor Position to X,Y }
Procedure SReadCursor(var X,Y: integer);{ Read Cursor Position }
Procedure SCurOn;                       { Enable display of Cursor }
Procedure SCurOff;                      { Disable display of Cursor }
Procedure SCurChr(C: char);             { Set cursor character }
Procedure SChrFunc(F: integer); { Set raster-op function for SPutChr }
Procedure SSetSize(Lines: integer; complemented, screenOff: Boolean);
                                { Set Screen Size; lines must be a
                                  multiple of 128; screenOff if true
                                  turns off display in part below lines
                                  in which case, complemented
                                  describes off part of screen }
Procedure CreateWindow(WIndx: WinRange;
                       OrgX, OrgY, Width, Height: integer; Title: string);
Procedure ChangeWindow(WIndx: WinRange);
Procedure GetWindowParms(var WIndx: WinRange;
          var OrgX, OrgY, Width, Height: integer; var hasTitle: Boolean);
Procedure ChangeTitle(Title: string);
Procedure SetFont(NewFont: FontPtr);
Function GetFont: FontPtr;
Procedure SClearChar(c: Char; funct: Integer); {delete prev char}
                                               { c BETTER NOT be CR or LF}
Procedure Line(Style: LineStyle; X1, Y1, X2, Y2: integer; Origin: RasterPt
Procedure SBackSpace(c: Char); {move back over last char of curLine}
                                               { c BETTER NOT be CR or LF}
Procedure RefreshWindow(WIndx: WinRange); {redraws window outline and titl
                                             area.  DOES NOT REDRAW TITLE}
Exception WBadSize; {parameter to SSetSize bad}
     Abstract:  Raised  if  the  lines  parameter to SSetSize is not a
          multiple of
               128 or is <=0.  Also raised  if  a  window  is  totally
               below  area to release so will disappear then if window
               # 0 or is the current window, then Raises WBadSize.
```

```
Exception BadWNum;  {indx is invalid}
```

Abstract: Raised if a window number parameter is illegal (not defined or out of range.

```
Exception WTooBig;
```

Abstract: Raised if parameters for new window specify an area that would extend off screen.

```
Procedure StartLine;
Procedure ToggleCursor;
Procedure NewLine;
Procedure SaveLineEnd(x: Integer);
Procedure SFullWindow;

Const SScreenW = 48;        {for use when want Screen in RasterOp or Line}
Var   SScreenP: RasterPtr; {for use when want Screen in RasterOp or Line}
      SCursorOn: boolean;
      SFunc: integer;       { Raster-op function for SPutChr }

{$ifc VarWin then}
      FirstWindp,             { first window's pointer; better not be NIL }
      CurWindp: WindowP;      { current window's pointer }
{$elsec}
      CurWind: WinRange;
      WinTable: Array[WinRange] of WindowType;
{$endc}

Exception CursOutSide;
```

Abstract: Raised if try to set the cursor outside of the current window.

Resume: Allowed. If resume, then cursor is NOT moved (same effect as if signal is caught but not resumed).

```
Procedure StartLine;
```

Abstract: Resets Curline and variables describing the current line start.

```
Procedure ToggleCursor;
```

Abstract: Inverts Cursor picture.

SideEffects: Changes the picture on the screen;

```
Procedure SSetCursor(x,y: integer);
```

Abstract: Moves the cursor to the specified screen position.

Parameters: x and y are Screen position where the next char will go. Note that y specified the BOTTOM of the character.

SideEffects: Changes the cur char positions AND sets line to be empty (so BS won't work);

Errors: Raises CursOutside if try to set the cursor outside the current window

```
Procedure SReadCursor(var x,y:integer);
```

Abstract: Returns the current screen coords for chars.

Parameters: x and y are set to the Screen position where the next char will go

```
Procedure SCurOn;
```

Abstract: Turns the char cursor on.

SideEffects: Changes SCursorOn global vble

```
Procedure SCurOff;
```

Abstract: Turns the char cursor off.

SideEffects: Changes SCursorOn global vble

Procedure SCurChr(C: char);

Abstract: Set the character to be used as the cursor.

SideEffects: Changes the cursor character

Procedure SChrFunc(F: integer);

Abstract: Set the function to be used for drawing chars to the screen.

SideEffects: Changes the char function

Procedure SSetSize(Lines: integer; complemented, screenOff: Boolean);

Abstract: Change the size of the screen so rest of memory can be used for other things (if smaller)

Parameters: Lines is the number of lines in the displayed part of the screen. It must be a multiple of 128 and > 0. Complemented describes the off part of the screen and screenOff determines whether it is displayed (false) or not; if displayed then complemented determines whether it is erased white or black.

Errors: if lines a bad value then Raises WBadSize. If a window is totally below area to release and will disappear then if window # 0 or is the current window, then Raises WBadSize.

SideEffects: Changes the values describing windows. If a window is totally below area to release and will disappear then if not window # 0 or is the current window, then makes the window undefined.

Procedure NewLine;

Abstract: Moves the cursor to the next line scrolling if necessary; DOES NOT do a CR

SideEffects: Changes the cursor position and may scroll

Procedure SaveLineEnd(x: Integer);

Abstract: Saves x as the end of a line

Parameters: x is the xPos of the end of a line

SideEffects: puts x at the end of LineEnds table; increments lastLineEnd; if table is full then scrolls table

```
Procedure SBackSpace(c: Char);
```

Abstract: Move the cursor back over c; c BETTER NOT be CR or LF

Parameters: c is the character to backspace over.

SideEffects: Moves the cursor back the width of char c; (DOES NOT ERASE CHAR)

```
Procedure SClearChar(c: char; funct: Integer);
```

Abstract: Deletes the c from screen; c BETTER NOT be CR or LF

Parameters: c is char to be erased; funct is RasterOp function to use in deleting char. It should be RXor if chars are black on white and RXNor if chars are white on black.

SideEffects: erases the last char of line;

```
Procedure SPutChr(CH: Char);
```

Abstract: Write a char into the current window

Parameters: Ch is char to write. If #200 bit is not set, checks to see if char is one of Bell, BS, FF, LF, CR and does something special.

SideEffects: Writes char to screen, moves cursor; may do a NewLine (and scroll) if at end of Line

```
Procedure ChangeTitle(Title: string);
```

Abstract: Changes the title of the current window (and displays new one).

Parameters: Title is new string. Characters in it are quoted so special characters will be displayed.

SideEffects: Changes title on screen

```
Procedure CreateWindow(WIndx: WinRange; OrgX,OrgY,Width,Height:
    integer; Title:string);
```

Abstract: Creates new window for Windx (or overwrites old values for that window) and makes it the current window. Writes title (IN CURRENT FONT) if title <> '';

Parameters: WIndx is index to use for the window created; OrgX and OrgY are the upper left corner of the outside of the new window (chars will be at least 5 bits in from that). Width and Height are total outside values for window (NOT the width and height of the character area). Title is title

for window. If not '' then hairlines and a black area are put around window.

SideEffects: Writes current values into current window; creates a new window and erases its area on screen

Errors: Raises BadWNum if WIndx invalid Raises WTooBig if window would extend off the screen

Procedure SFullWindow;

Abstract: Changes the parameters of the current window to be the full screen

SideEffects: Changes the size of the current window. Does NOT refresh or change the title line or erase anything or move the cursor

Procedure RefreshWindow(WIndx: WinRange);

Abstract: Redraws window outline and title area (but not title text)

Parameters: Window to refresh (better be already created)

Errors: Raises BadWNum if WIndx undefined

Procedure GetWindowParms(var WIndx: WinRange; var OrgX, OrgY, Width, Height: integer; var hasTitle: Boolean);

Abstract: Returns parameters for current window

Parameters: All set to current window's values

Procedure ChangeWindow(WIndx: WinRange);

Abstract: Writes out current window's parameters and changes to new one

Parameters: WindX is new window's number

Errors: Raises BadWNum if WIndx undefined

```
Procedure SetFont(NewFont: FontPtr);
```

Abstract: Changes font to be NewFont

Parameters: NewFont is font to use

SideEffects: Changes font in current window so all further writes (including titles) will be in this font

```
Function GetFont: FontPtr;
```

Abstract: Returns current font

Returns: font currently in use

```
Procedure ScreenReset;
```

Abstract: Erases screen; Removes all window; sets Window 0 to have full screen boundary and a blank title

SideEffects: Erases or sets all parameters; font set to system font

```
Procedure ScreenInit;
```

Abstract: Sets FirstWindP to NIL and sets up default window; NOTE: CALL THIS PROCEDURE ONCE AT SYSTEM INITIALIZE

Calls: ScreenReset;

```
Procedure Line(Style: LineStyle; X1, Y1, X2, Y2: integer; Origin:
    RasterPtr);
```

Abstract: Draws a line.

Parameters: Style is function for the line; X1, X2, Y1, Y2 are end points of line, Origin is pointer to the memory to draw lines in. Use SScreenP for Origin to draw lines on the screen.

```
Module Scrounge;

Abstract: This module contains the procedure Scrounge which allows a
small
     amount of debugging.  Since there are no symbol tables or micro--
     code support for breakpoints, you can look at the stack trace and
     examine  vbles by offsets.  The types of the variables have to be
     specified by the user.  Written by: Brad  A.  Myers  1-May-1981
     Copyright (C) 1981 Three Rivers Computer Corporation.

Version Number V0.14
{//////////////////////////} EXPORTS {\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\}

Procedure Scrounge(ES, ER, PStart, PEnd, ExcSeg, RaiseAP: Integer);
```

```
Procedure Scrounge(ES, ER, PStart, PEnd, ExcSeg, RaiseAP: Integer);
```

Abstract: Scrounge is called when uncaught signals are noticed or when the user types ^SHIFT-D. It allows looking around at local and global vbles and the stack trace. If ^SHIFT-D then can continue with program, otherwise aborts when exit

Parameters: ES - segment number of exception
ER - routine number of exception
PStart - offset of start of parameters to exception
PEnd - offset of end of parameters to exception
ExcSeg - the segment number of the exceptions module if (ES = ExcSeg) and (ER = ErrDump) then is ^SHIFT-D For now, can't tell ^SHIFT-C
RaiseAP - the offset for AP for Raise itself (caller is person who did the raise)

```
'dule Stream;


;ream - Perq Pascal stream package.
)hn Strait                ca.  Jan 80.
)pyright (C) Three Rivers Computer Corporation, 1980.


)stract:
    This module implements the low-level Pascal I/O.  It is not
    intended for use directly by user programs, but rather the
    compiler generates calls to these routines when a Reset, Rewrite,
    Get, or Put is encountered.  Higher-level character I/O functions
    (Read and Write) are implemented by the two modules Reader and
    Writer.

    In this module, the term "file buffer variable" refers to F^ for
    a file variable F.

'rsion Number V1.20
'ports


'ports FileDefs from FileDefs;


  t StreamVersion = '1.20';

    IdentLength = 8;          { significant characters in an identifier }


'pe pStreamBuffer = ^StreamBuffer;

    StreamBuffer = record case integer of  { element size: }
     0: (W: array[0..255] of integer);       { 1 or more words, or > 8 bits }
     1: (B1: packed array[0..0] of 0..1);    { 1 bit }
     2: (B2: packed array[0..0] of 0..3);    { 2 bits }
     3: (B3: packed array[0..0] of 0..7);    { 3 bits }
     4: (B4: packed array[0..0] of 0..15);   { 4 bits }
     5: (B5: packed array[0..0] of 0..31);   { 5 bits }
     6: (B6: packed array[0..0] of 0..63);   { 6 bits }
     7: (B7: packed array[0..0] of 0..127);  { 7 bits }
     8: (B8: packed array[0..0] of 0..255);  { 8 bits }
     9: (C: packed array[0..255] of char);   { for character structured }
     end;


    ControlChar = 0..#37;                  { ordinal of an ASCII control
                                             character }

    FileKind = (BlockStructured, CharacterStructured);

    FileType = { file of Thing }
     packed record
      Flag: packed record case integer of
       0: (CharReady  : boolean;            { character is in file window }
```

```
        FEoln       : boolean;          { end of line flag }
        FEof        : boolean;          { end of file }
        FNotReset   : boolean;          { false if a Reset has been
                                          performed on this file }
        FNotOpen    : boolean;          { false if file is open }
        FNotRewrite: boolean;           { set false if a Rewrite has been
                                          performed on this file }
        FExternal   : boolean;          { not used - will be permanent/temp
                                          file flag }
        FBusy       : boolean;          { IO is in progress }
        FKind       : FileKind);
    1: (skip1      : 0..3;
        ReadError  : 0..7);
    2: (skip2      : 0..15;
        WriteError: 0..3)
    end;
  EolCh, EofCh, EraseCh, NoiseCh: ControlChar; {self explanatory}
  OmitCh          : set of ControlChar;
  FileNum         : integer;        { POS file number }
  Index           : integer;        { current word in buffer for un-packed
                                      files, current element for packed
                                      files }
  Length          : integer;        { length of buffer in words for un-
                                      packed files, in elements for packed
                                      files }
  BlockNumber     : integer;        { next logical block number }
  Buffer          : pStreamBuffer;{ I/O buffer }
  LengthInBlocks: integer;          { file length in blocks }
  LastBlockLength:integer;          { last block length in bits }
  SizeInWords     : integer;        { element size in words, 0 means
                                      packed file }
  SizeInBits      : 0..16;          { element size in bits for packed
                                      files }
  ElsPerWord      : 0..16;           { elements per word for packed files }
  Element: { Thing } record case integer of  {The File window}
    1: (C: char);
    2: (W: array[0..0] of integer)
    end
  end;


ChArray = packed array[1..1] of char; {For read/write character array}


Identifier = string[IdentLength];
IdentTable = array[0..1] of Identifier;



var StreamSegment: integer;     { Segment buffer for I/O buffers }
    KeyBuffer: packed array[0..255] of char;
    KeyNext, KeyLength: integer;
```

```
procedure StreamInit( var F: FileType; WordSize, BitSize: integer;
                      CharFile: boolean );
procedure StreamOpen( var F: FileType; var Name: PathName;
                      WordSize, BitSize: integer; CharFile: boolean;
                      OpenWrite: boolean );
procedure StreamClose( var F: FileType );
procedure GetB( var F: Filetype );
procedure PutB( var F: Filetype );
procedure GetC( var F: Filetype );
procedure PutC( var F: FileType );
procedure PReadln( var F: Filetype );
procedure PWriteln( var F: Filetype );
procedure InitStream;
function StreamName( var F: FileType ): PathName;
function FullLn( var F: Text ): Boolean;
procedure StreamKeyBoardReset( var F: Text );


exception ResetError( FileName: PathName );
```

Abstract:
Raised when unable to reset a file--usually file not found but also could be ill-formatted name or bad device name.

Parameters:
FileName - name of the file or device.

```
exception RewriteError( FileName: PathName );
```

Abstract:
Raised when unable to rewrite a file--usually file unknown device or partition but also could be ill-formatted name or bad device name.

Parameters:
FileName - name of the file or device.

```
exception NotTextFile( FileName: PathName );
```

Abstract:
Raised when an attempt is made to open a non-text file to a character-structured device.

Parameters:
FileName - name of the device.

```
exception NotOpen;
```

Abstract:
Raised when an attempt is made to use a file which is not open.

```
exception NotReset( FileName: PathName );
```

Abstract:
Raised when an attempt is made to read a file which is open but has not been reset.

Parameters:
FileName - name of the file or device.

```
exception NotRewrite( FileName: PathName );
```

Abstract:
Raised when an attempt is made to write a file which is open but has not been rewritten.

Parameters:
FileName - name of the file or device.

```
exception PastEof( FileName: PathName );
```

Abstract:
Raised when an attempt is made to read past the end of the file.

Parameters:
FileName - name of the file or device.

```
exception UnitIOError( FileName: PathName );
```

Abstract:
Raised when IOCRead or IOCWrite returns an error status.

Parameters:
FileName - name of the device.

```
exception TimeOutError( FileName: PathName );
```

Abstract:
Raised when a device times out.

Parameters:
FileName - name of the device.

```
exception UndfDevice;
```

Abstract:
Raised when an attempt is made to reference a file which is open to a character-structured device, but the device number is bad. In the current system (lacking automatic initialization of file variables), this may be caused by referencing a file which has never been opened.

```
exception NotIdentifier( FileName: PathName );
```

Abstract:
Raised when an identifier is expected on a file, but something else is encountered.

Parameters:
FileName - name of the file or device.

```
exception NotBoolean( FileName: PathName );
```

Abstract:
Raised when a boolean is expected on a file, but something else is encountered.

Parameters:
FileName - name of the file or device.

```
exception BadIdTable( FileName: PathName );
```

Abstract:
Raised by ReadIdentifier when the identifier table is bad.

Parameters:
FileName - name of the file or device.

```
exception IdNotUnique( FileName: PathName; Id: Identifier );
```

Abstract:
Raised when non-unique identifier is read.

Parameters:
FileName - name of the file or device. Id - the identifier which was read.

```
exception IdNotDefined( FileName: PathName; Id: Identifier );
```

Abstract:
Raised when an undefined identifier is read.

Parameters:
FileName - name of the file or device. Id - the identifier which was read.

```
exception NotNumber( FileName: PathName );
```

Abstract:
Raised when a number is expected on a file, but something else is encountered.

Parameters:
FileName - name of the file or device.

```
exception LargeNumber( FileName: PathName );
```

Abstract:
Raised when a number is read from a file, but it is too large.

Parameters:
FileName - name of the file or device.

```
exception SmallReal( FileName: PathName );
```

Abstract:
Raised when a real number is read from a file, but it is too small.

Parameters:
FileName - name of the file or device.

```
exception BadBase( FileName: PathName; Base: Integer );
```

Abstract:
Raised when an attempt is made to read a number with a numeric base that is not in the range 2..36.

Parameters:
FileName - name of the file or device. Base - numeric base (which is not in the range 2..36).

```
exception LargeReal( FileName: PathName );
```

Abstract:
Raised when a real number is read from a file, but it is too large.

Parameters:
FileName - name of the file or device.

```
exception RealWriteError( FileName: PathName );
```

Abstract:
Raised when an attempt is made to write a real number which is invalid.

Parameters:
FileName - name of the file or device.

```
exception NotReal( FileName: PathName );
```

Abstract:
Raised when a real number is expected on a file, but something else is encountered.

Parameters:
FileName - name of the file or device.

```
procedure StreamInit( var F: FileType; WordSize, BitSize: integer;
     CharFile: boolean );
```

Abstract: Initializes, but does not open, the file variable F. Automatically called upon entry to the block in which the file is declared. (To be written when the compiler generates calls to it.)

Parameters:
F - the file variable to be initialized. WordSize and BitSize are the size of an element of the file.
CharFile - determines whether or not the file is of characters.

```
procedure StreamClose( var F: FileType );
```

Abstract: Closes the file variable F.

Parameters:
F - the file variable to be closed.

```
procedure StreamOpen( var F: FileType; var Name: PathName; WordSize,
     BitSize: integer; CharFile: boolean; OpenWrite: boolean );
```

Abstract: Opens the file variable F. This procedure corresponds to both Reset and Rewrite.

Parameters:
F - the file variable to be opened.
Name - the file name.
WordSize - number of words in an element of the file (0 indicates a packed file).
BitSize - number of bits in an element of the file (for packed files).
CharFile - true if the file is a character file.
OpenWrite - true if the file is to be opened for writing (otherwise it is opened for reading).

Errors: ResetError if unable to reset the file. RewriteError if unable to rewrite the file. NotATextFile if an attempt is made to open a non-text file to a character structured device.

```
procedure GetB( var F: Filetype );
```

Abstract: Advances to the next element of a block-structured file and gets it into the file buffer variable.

Parameters:
F - the file to be advanced.

Errors: NotOpen if F is not open. NotReset if F has not been reset. PastEof if an attempt is made to read F past Eof.

```
procedure GetC( var F: Filetype );
```

Abstract: Advances to the next element of a character-structured file and gets it into the file buffer variable.

Parameters:
F - the file to be advanced.

Errors:
NotOpen - if F is not open.
NotReset - if F has not been reset.
PastEof - if an attempt is made to read F past Eof.
TimeOutError - if RS: or RSX: times out.
UnitIOError - if IOCRead doesn't return IOEIOC or IOEIOB.
UndfDevice - if F is open, but the device number is bad.

```
procedure PutB( var F: Filetype );
```

Abstract: Writes the value of the file buffer variable to the block-structured file and advances the file.

Parameters:
F - the file to be advanced.

Errors:
NotOpen - if F is not open.
NotRewrite- if F has not been rewritten.

```
procedure PutC( var F: FileType );
```

Abstract: Writes the value of the file buffer variable to the character- structured file and advances the file.

Parameters:
F - the file to be advanced.

Errors:
NotOpen - if F is not open.
NotRewrite - if F has not been rewritten.
UnitIOError- if IOCWrite doesn't return IOEIOC or IOEIOB.
TimeOutError- if RS: or RSX: times out.
UndfDevice - if F is open, but the device number is bad.

```
procedure PReadln( var F: Filetype );
```

Abstract: Advances to the first character following an end-of-line.

Parameters:
F - the file to be advanced.

```
procedure PWriteln( var F: Filetype );
```

Abstract: Writes an end-of-line.

Parameters:
F - the file to which an end-of-line is written.

```
procedure StreamKeyBoardReset( var F: Text );
```

Abstract: Clears the keyboard input buffer and the file variable F so that all input typed up to this point will be ignored.

Parameters:
F - file to be cleared.

```
procedure InitStream;
```

Abstract: Initializes the stream package. Called by System.

```
function FullLn( var F: Text ): Boolean;
```

Abstract: Determines if there is a full line in the keyboard input buffer. This is the case if a carriage-return has been typed. This function is provided in order that a program may continue to do other things while waiting for keyboard input. If the file is not open to the console, FullLn is always true.

Parameters:
F - file to be checked.

Returns: True if a full line has been typed.

Errors:
NotOpen - if F is not open.
NotReset - if F has not been reset.

```
function StreamName( var F: FileType ): PathName;
```

Abstract: Returns the file name associated with the file variable F. For block-structured files, the full path name including device and partition is returned. For character-structured files, the device name is returned.

Parameters:
F - file variable whose name is to be returned.

```
OS D.6 Interface                                        Program System
5 Feb 82

'ogram System;


erq Software Group.
opyright (C) Three Rivers Computer Corporation, 1980, 1981. .

ostract:
    Initialize POS and go into loop  alternately  running  Shell  and
    user program
ersion Number V2.2
**********} Exports {**********}

onst MainVersion = 'D';
     DebugSystemInit = False;
     FirstDDS = 199;
     ShellConst = 'Shell.';
     LogConst = 'LogIn.';
     PFileConst = 'Default.Profile';

     SysTiming = True;     { Gather System timing statistics.  If this constant
                             is changed, IO, Loader, Memory, Movemem, System,
                             and Shell should be re-compiled, and the System
                             should be re-linked.  }


me Sys9s = String[10];


ar UsrCmdLine: String[255];     {Command line entered by user}
   UseCmd: Boolean;             {Set True to tell shell to execute UsrCmdLine}
   InCmdFile: Boolean;          {True if shell commands from file}
   LastFileName,                {Name of file to use if none given}
   RFileName,                   {Name of next program to run}
   ShellName: String;           {Name of Shell}

   CurUserID,                   {Index of user in System.Users}
   CurGroupID: 0..255;          {Groupid of current user}
   CurUserName,                 {LogIn name of current user}
   CurPFile: String;            {Name of current profile file}
   UserMode: Boolean;           {True while executing user program}

   CtrlCPending: Boolean;       {True if one control-C typed}

   NextSSize: Integer;          {Screen size for next program}
   NextSComplemented: Boolean;  {Whether to complement bottom for next pgm}
   NextSOff: Boolean;           {Whether bottom should display data bits}
   DefCursFunct: Integer;       {What to set curs func to after each prog}
   DefScrComp: Boolean;         {Default value for NextSComplemented}
   DefScrOff: Boolean;          {Default value for NextSOff}

   ShellCtrl: pointer;          {Pointer to information record for Shell}
   TimeFID: integer;            {File ID of file holding current time}
   CmdSegment: Integer;         {SegmentNumber of seg holding command files}

   InPmd: Boolean;              {True if in Scrounge (PostMortemDump)}
   SysDisk: Integer;            {Number of the disk booted from}
```

```
    SysBootChar: Integer;       {Ord(char held down to boot)}

    StrVersion: string;         {System version number as a string}
    SystemVersion: Integer;     {Integer giving system version number}
    SystemInitialized: Boolean; {True after system initialized}
    DDS: Integer;               {Keeps current diagnostic display value}
    ShouldReEnableSwapping: Boolean;  {True if swapping must be reenabled}
    SavedSwapId: Integer;       {Save id of where to swap to}

    {$ifc SysTiming then}
    LoadTime, OldLoadTime: long;
    ExecuteTime, OldExecuteTime: long;
    SwapTime, OldSwapTime: long;
    MoveTime, OldMoveTime: long;
    IOTime, OldIOTime: long;
    PrintStatistics: Boolean;
    {$endc}


    UserPtr: pointer;           {A pointer variable for use between user
                                 programs.  (Use IncRefCount to keep segment)}
    UserInt: integer;           {May be a segment number for UserPtr}

    DemoInt: Integer;           {reserved for Demo system}

    isFloppy: Boolean;          {true if booted from floppy, else false}
    pointAllowed: Boolean;      {true if should use pointing device}


{*** WARNING!!  IF YOU CHANGE THE EXPORTED PROCEDURES AND EXCEPTIONS, MAKE
{***            SURE THE NUMBERS FOR THE FOLLOWING EXCEPTIONS ARE UPDATED
{***            AND RECOMPILE SCROUNGE IF CHANGED !!!!! *****}

{*** WARNING!!  DO NOT CHANGE THE ORDER OF THE ^C EXCEPTIONS !!!!! *****}

Procedure Command;
Procedure SetDDS( Display: Integer );
Procedure SysVers( n: integer; var S: string );

Const ErrCtlC = 4; {******}
Exception CtlC;


     Abstract:
               CtlC is raised by the KeyBoard interrupt routine when a
               control-c is typed. If you handle this exception you
               should clear CtrlCPending in your handler. If you are
               catching control-c's to try to prevent aborts, you
               should enable CtlCAbort also, since the Stream package
               will raise it when the control-c is read.


Const ErrCtlCAbort = 5; {******}
Exception CtlCAbort;
```

Abstract:

CtlCAbort is raised by the KeyBoard interrupt routine when the second of two adjacent control-c's is typed. It is also raised by the Stream package when a control-c is read. If you handle this exception you should clear CtrlCPending in your handler.

When this is raised by the KeyBoard interrupt routine, the KeyBoard type-ahead buffer is cleared. If you want to prevent this, you must catch CtlC also.

If your program uses a Text file and you want to clear the line editing buffer for that file, you should call the Stream routine StreamKeyBoardReset(F) (assuming F is the name of the file). If F is a Text file which is attached to the console, this will get rid of the character F^ points to and clear Stream's line editing buffer.

```
Const ErrCtlShftC = 6; {******}
Exception CtlShftC;
```

Abstract:

CtlShftC is raised by the KeyBoard interrupt routine when a control- shift-c is typed. If you handle this exception you should clear CtrlCPending in your handler.

When this is raised by the KeyBoard interrupt routine, the KeyBoard type-ahead buffer is cleared. You cannot prevent this.

If your program uses a Text file and you want to clear the line editing buffer for that file, you should call the Stream routine StreamKeyBoardReset(F) (assuming F is the name of the file). If F is a Text file which is attached to the console, this will get rid of the character F^ points to and clear Stream's line editing buffer.

```
Const ErrExitProgram = 7; {******}
Exception ExitProgram;
```

Abstract:

ExitProgram is raised to abort (or exit) a program. The default handler for CtlCAbort and Scrounge raise this exception.

WARNING: No one but System and Loader should Handle this exception. Anyone may raise it to exit a program.

```
Const ErrHelpKey = 8; {******}
Exception HelpKey(var retStr: Sys9s);
```

Abstract:

HelpKey is raised when the HELP key is hit.

Parameters:

retStr - the set of characters to put into the input stream. This should be set by the handler if it continues from the exception. Likely values are "/Help<CR>" and chr(7) (the current value returned). The key board interupt routine sets retStr to '' before raising this exception so if not set, and the handler resumes, nothing will be put into the input stream.

Resume:

Allowed. Should set retStr first.

```
type DoubleWord = ^integer;    {should use Long instead}
```

```
Procedure SetDDS( Display: Integer );
```

Abstract: SetDDS sets the diagnostic display to a particular value.

Parameters:
Display - Desired value of the diagnostic display.

```
procedure SysVers( n: integer; var S: string );
```

Abstract: This procedure will provide the caller with a string that is the version number of the current system.

Parameters: n is the minor version number of the system.

S will be set to the current minor version of the system.

```
Procedure Command;
```

Abstract: This procedure alternately loads Shell and the user programs whose runfile names are generated by Shell. It is invoked by the main program in System and can be exited only if the user types ^C or if a runtime error occurs.

```
Module SystemDefs;


SystemDefs - Common system definitions.
John P.  Strait  13 May 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     SystemDefs exports common system Const and Type definitions.  The
     intent  is  that  SystemDefs should not export Procedures or Vars
     since these require  a  Seg  file.   It  is  also  intended  that
     SystemDefs  be  reasonably  short so that it doesn't take long to
     import.

Version Number V1.2
exports


const Ether3MBaud = False;              { no support for 3 MBaud EtherNet }
      Ether10MBaud = True;             { no support for 10 MBaud EtherNet }


type Double = array[0..1] of Integer;
```

```
)S D.6 Interface                                        module UserPass
5 Feb 82

'ule UserPass;


)stract:
    This module provides facilities for dealing with the password and
    accounts file for PERQ.  The login and protection facilities  for
    Perq  provide  a very simple user validification.  This system is
    NOT completly secure.

    Written by: Don Scelza

    Copyright (C) Three Rivers Computer Corporation,  1981.

ersion Number V1.3
**********} Exports {**********}

/pe IDType  = 0..255;

   PassType =  ^Integer;                { a two word value }

   UserRecord = packed record
           InUse: boolean;                  { is this entry in use.  }
           Name: String[31];                { Name of the user }
           UserID: IDType;                  { The user ID of the user.  }
           GroupID: IDType;                 { The group ID of the user.  }
           EncryptPass: PassType;           { The encrypted password.  }
           Profile: String;                 { Path name of the profile file.  }
       end;


)nction FindUser(UserName: String;  var UserRec: UserRecord): Boolean;
)nction ValidUser( UserName, Password: String;  var UserRec: UserRecord): Boole
)nction AddUser(UserName, Password: String;  Group: IDType;
                ProPath: String): Boolean;
'ocedure NewUserFile;
'ocedure ListUsers;
)nction RemoveUser(UserName: String): boolean;

)nst PassFile = '>System.Users';

)nst MaxUsers = 10;

'pe Users = array[0..MaxUsers] of UserRecord;
```

function FindUser(UserName: String; var UserRec: UserRecord): Boolean;

Abstract: This function is used to see if a user exists in the user file.

Parameters: UserName is the name of the user that we are looking for.

UserRec is a var parameter that is used to return the information about the user UserName if he is in the file.

Results: This procedure will return true if the user UserName was in the user file. It will return False otherwise.

function ValidUser(UserName, Password: String; var UserRec: UserRecord): Boole

Abstract: This function is used to see if a user name and password match.

Parameters: Username is the name of the user that we want to check.

Password is the password for the user.

UserRec will be filled with the user information if the user name and password match.

Results: If the password is valid for the user then return true. Otherwise return false.

Side Effects: This function will change the file PassFile.

function AddUser(UserName, Password: String; Group: IDType; ProPath: String): Boolean;

Abstract: This function is used to add a new user to the user file or change the parameters of an already existing user.

Parameters: Username is the name of the user that we want to add or change.

Password is the password for the user.

Group is the group number for the new user.

ProPath is the path name of the profile file for this user.

Results: If the user could be added or changed then return true. Otherwise return false.

Side Effects: This function will change the file PassFile.

```
procedure NewUserFile;
```

Abstract: This procedure is used to create a new user file.

Side Effects: This procedure will create a new file. It will destroy any information in the current file.

```
procedure ListUsers;
```

Abstract: This procedure is used to supply a list of the valid users.

```
function RemoveUser(UserName: String): boolean;
```

Abstract: This procedure is usd to remove a user from the list of valid users.

Parameters: UserName is the name of the user that is to be removed.

Results: If the user could be removed the return true. Otherwise return false.

```
Module UtilProgress;


Progress Reporting Routines
Copyright (C) 1981  Three Rivers Computer Corporation

Abstract:
     Routines to show progress of utilities.

Exports
Procedure LoadCurs;
Procedure ShowProgress(NumLines: Integer);
Procedure QuitProgress;
Procedure StreamProgress( var F: File );
Procedure ComputeProgress( Current, Max: Integer );
Procedure LoadBusy;
```

Procedure LoadCurs;

Abstract: Sets up the cursor before showing progress.

Procedure LoadBusy;

Abstract: Sets up the cursor so that we can show that we are busy. In busy mode, each ShowProgress moves the cursor by one in a random direction. This should be used when an operation is taking place and the utility cannot tell how long until it is done.

Procedure QuitProgress;

Abstract: No more progress to report, turn off the cursor.

Calls: IOCursorMode.

Procedure ShowProgress(NumLines: Integer);

Abstract: If started by LoadCurs then Indicate progress by moving the cursor down a certain number of scan lines. If started by LoadBusy then update busy cursor to show that doing something.

Parameters: NumLines is the number of scan lines to move the cursor.

Side Effects: CursPos is modified. BusyX is modified if <> -1.

Environment: Assumes LoadCurs or LoadBusy has been called.

Calls: IOSetCursorPos.

Procedure StreamProgress(var F: File);

Abstract: Indicate progress reading a Stream file.

Parameters: F is a Stream file which has been Reset.

Side Effects: CursPos is modified.

Calls: IOSetCursorPos.

Errors: NotOpen if F is not open. NotReset if F is open but not Reset.

Procedure ComputeProgress(Current, Max: Integer);

Abstract: Indicate progress given a current and maximum value.

Parameters: Current is the current value. Max is the maximum value.

Side Effects: CursPos is modified.

Calls: IOSetCursorPos.

```
module Virtual;


Virtual - Perq virtual memory manager.
J. P. Strait          1 Jan 80.
Copyright (C) Three Rivers Computer Corporation, 1980.

Abstract:
     Virtual is the Perq virtual memory manager.  It supervises the
     segment tables and exports procedures for swapping memory
     segments. Virtual is the portion of the Perq memory manager
     which must remain memory resident at all times. Perq physical
     memory is segmented into separately swappable items (called
     segments) which may contain either code or data.

     Design: See the Q-Code reference manual.

Version Number V2.8
exports


const VirtualVersion = '2.8';


imports Memory from Memory;
imports IO_Unit from IO_Unit;
imports DiskIO from DiskIO;


function  ReturnSegment: SegmentNumber;
procedure ReleaseSegmentNumber( Seg: SegmentNumber );
function  NewSegmentNumber: SegmentNumber;
procedure MakeEdge( var E: MMEdge; S: SegmentNumber );
procedure DeleteSegment( var S: SegmentNumber );
procedure SwapOut( var E: MMEdge );
procedure SwapIn( E: MMEdge; S: SegmentNumber; P: MMPosition );
procedure Compact;
procedure KeepSegments;
procedure FindHole( Fsize: MMIntSize; ForUserSegment: Boolean );
procedure IncIOCount( S: SegmentNumber );
procedure DecIOCount( S: SegmentNumber );
procedure SwapSegmentsIn( S1, S2, S3, S4: SegmentNumber );



var ScreenLast: Integer;
    Keep1, Keep2, Keep3, Keep4: SegmentNumber;
    Kludge: record case Integer of
              1: (A: DiskAddress);
              2: (D: Double)
              end;
    BlockHeader: IOHeadPtr;
    BlockAddress: Double;
    BlockSId: SegId;
    Status: IOStatPtr;
    BootSerialNum: Double;
```

```
    BootSegId: SegId;
    SwapSId: SegId;
```

```
function ReturnSegment: SegmentNumber;
```

Abstract: ReturnSegment finds the segment number of the caller of the procedure which called ReturnSegment by searching the call stack.

Result: ReturnSegment = Segment number of the caller.

Design: This routine depends on the Perq running a single process operating system where the caller is in the same process as the memory manager.

```
procedure ReleaseSegmentNumber( Seg: SegmentNumber );
```

Abstract: ReleaseSegmentNumber releases a segment number to the list of segment numbers which are not in use.

Parameters:
Seg - Segment number to return to the segment number free list.

```
function NewSegmentNumber: SegmentNumber;
```

Abstract: NewSegmentNumber allocates the next unused segment number.

Errors: NoFreeSegments if there are no unused segment numbers.

```
procedure MakeEdge( var E: MMEdge; S: SegmentNumber );
```

Abstract: MakeEdge makes an MMEdge record which the head field set to a certain segment number and the tail field set to the previous segment number (in physical address order).

Parameters:
E - MMEdge record to build.
S - Segment to put in the head field.

Errors: EdgeFailure if MakeEdge can't find the previous segment number.

```
procedure DeleteSegment( var S: SegmentNumber );
```

Abstract: DeleteSegment returns a segment to the free memory list. This is done (for example) when the segment's reference and IO counts both reach zero.

Parameters:

S - Number of the segment to be destroyed. To facilitate segment table scanning loops that contain calls to DeleteSegment: * If S was resident, it is changed to be the number of the segment which represents the free memory. This may not be the same as the original value if the original segment is coalesced with an adjacent free segment. * If S was not resident, it is changed to be the number of the segment which preceded it in the segment table. * MMFirst is set to have the same value as S on exit.

```
procedure SwapOut( var E: MMEdge );
```

Abstract: SwapOut swaps a data segment out to disk.

Parameters:

E - An edge where the head is the segment to be swapped and the tail is the previous segment.

Result: E.T and E.H both are set to the number of the new free segment.

Errors: PartNotMounted if the swapping partition is not mounted. SwapError if attempt to swap segment out while swapping is disabled.

```
procedure SwapIn( E: MMEdge; S: SegmentNumber; P: MMPosition );
```

Abstract: SwapIn swaps a segment in from disk.

Parameters:

- E - An edge describing where to put the segment in memory. The head is a free segment which will be filled by the segment to be swapped in. The tail is the previous segment.
- S - The segment to swap in.
- P - The position (low end or high end) to use within the head segment of the edge.

Errors: SwapInFailure if attempt to swap in a segment which was never swapped out.

```
procedure Compact;
```

Abstract: Compact compacts physical memory by moving as many segments as possible toward low addresses. System segments (those with a reference count greater than one) will not be moved into the screen area, as segments cannot jump over one another.

Errors: CantMoveSegment if attempt to move a segment with non-zero IO count.

```
procedure KeepSegments;
```

Abstract: KeepSegments marks the segments Keep1 through Keep4 as not RecentlyUsed so that they won't be swapped out.

```
procedure FindHole( Fsize: MMIntSize; ForUserSegment: Boolean );
```

Abstract: FindHole attempts to find a hole (free memory) of a certain size. It performs a first-fit search. If a hole cannot be found, memory is compacted, and another first-fit search is performed. Eventually, a swap-out pass will be performed.

Parameters:
Fsize - Minimum size of the hole. This is an internal size--Fsize=n means n+1 blocks.
ForUserSegment - True iff this hole is to be used for a user segment. System segments may not be allocated in the screen area.

```
procedure IncIOCount( S: SegmentNumber );
```

Abstract: IncIOCount increments the count of input/output references to a data segment. A non-zero IO count prevents a segment from being moved, swapped, or destroyed.

Parameters:
S - Segment number.

Errors: UnusedSegment if S is not in use. FullMemory if S is not resident and there isn't enough memory to swap it in.

```
procedure DecIOCount( S: SegmentNumber );
```

Abstract: DecIOCount decrements the IO count of a data segment by one. If the reference and IO counts both become zero: * if the segment is a data segment, it is destroyed. * if the segment is a code segment, it is destroyed only if it is in the screen or is non-resident.

Parameters:
S - Number of the segment.

Errors: UnusedSegment if S is not in use.

```
procedure SwapSegmentsIn( S1, S2, S3, S4: SegmentNumber );
```

Abstract: SwapSegmentsIn ensures that when it returns, S1, S2, S3, and S4 are resident.

Parameters: S1, S2, S3, S4 - segments to swap in.

Errors: NilPointer if one of the segments is zero.
UnusedSegment if one of the segments is not really in use.
FullMemory if there isn't enough memory to swap one of the segments in.

```
module Writer;


Writer - Stream package output conversion routines.
J.  P.  Strait   ca.  1 Jan 81.
Copyright (C) Three Rivers Computer Corporation, 1981.

Abstract:
     Writer is the character output module of the Stream package.   It
     is called by code generated by the Pascal compiler in response to
     a Write or Writeln.  It is one level above Module Stream and uses
     Stream's output routines.

Version Number V2.2
exports


imports Stream from Stream;


procedure WriteBoolean( var F: FileType; X: Boolean; Field: integer );
procedure WriteCh( Var F: FileType; X: char; Field: integer );
procedure WriteChArray( var F: FileType; var X: ChArray; Max, Field: integ
procedure WriteIdentifier( var F: FileType; X: integer;
                           var IT: IdentTable; L, Field: integer );
procedure WriteInteger( var F: FileType; X: integer; Field: integer );
procedure WriteString( var F: FileType; var X: String; Field: integer );
procedure WriteX( var F: FileType; X, Field, B: integer );
```

```
procedure WriteBoolean( var F: FileType; X: Boolean; Field: integer );
```

Abstract: Writes a boolean in fixed format.

Parameters:
X - the boolean to be written.
F - the file into which X is to be written.
Field - the size of the field into which X is to be written.

```
procedure WriteCh( var F: FileType; X: char; Field: integer );
```

Abstract: Writes a character in a fixed format.

Parameters:
X - the character to be written.
F - the file into which X is to be written.
Field - the size of the field into which X is to be written.

```
procedure WriteChArray( var F: FileType; var X: ChArray;  Max,  Field:
    integer );
```

Abstract: Writes a packed character array in fixed format.

Parameters:
X - the character array to be written.
F - the file into which X is to be written.
Field - the size of the field into which X is to be written.
Max - the declared length of X.

```
procedure WriteIdentifier( var F: FileType; X: integer; var IT:
    IdentTable; L, Field: integer );
```

Abstract:

Writes an identifier from a table in fixed format.

Parameters:

X - the ordinal of the identifier in the range 0 to L.
F - the file to which X is written.
IT - the table of identifier names indexed from 0 to L.

L - the largest identifier ordinal defined by the table.
Field - the size of the field into which X is written.

Errors: BadIdTable if the length of the identifier table is less than 1.

```
procedure WriteInteger( var F: FileType; X: integer; Field: integer );
```

Abstract: Writes a decimal integer in fixed format.

Parameters:
X - the integer to be written.
F - the file into which X is to be written.
Field - the size of the field into which X is to be written.

```
procedure WriteString( var F: FileType; var X: String; Field: integer
    );
```

Abstract: Writes a string in fixed format.

Parameters:
X - the string to be written.
F - the file into which X is written.
Field - the size of the field into which X is written.

```
procedure WriteX( var F: FileType; X, Field, B: integer );
```

Abstract: Writes an integer in fixed format with base B.

Parameters:
X - the integer to be written.
F - the file into which X is to be written.
Field - the size of the field into which X is to be written.
B - the base of X. It is an integer whose absolute value must be between 2 and 36, inclusive.

Errors: BadBase if the base is not in 2..36.

Programming Examples

Brad A. Myers

This manual describes how to use Fonts, Cursors, RasterOp, Line, Windows, CmdParse, PopUp menus, and how to allocate large amounts of memory. The manual includes examples of sample applications.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

1. Introduction.

This manual describes how to use the PERQ operating system to perform some interesting operations. Examples are given of the ways we have found to be successful in performing these operations. Although there are obviously many ways to perform these operations, the ones given here are successful.

2. Allocating Memory.

This section describes how to use the Memory module to allocate blocks of memory needed for reading in Fonts and pictures from files, for creating pictures off screen for RasterOp, and for handling large amounts of data.

Fonts and pictures are generally stored in files on the disk. To use the fonts and pictures, they must first be read into memory. First, do a FSLookUp (or one of the other lookup functions) from FileSystem. A VAR parameter to this function is the number of blocks in the file. This number can be passed to the memory manager to tell it how much storage to allocate. Memory on the PERQ is divided into "segments". Each segment can have up to 256 blocks. Each block is 256 words or 512 bytes. When a segment is created, it is given an initial size, a maximum size, and an increment by which to increase the current size when that is not enough. A segment is created by using the procedure from memory:

```
Procedure CreateSegment(
            var Seg: Integer;
            initialSize,              {in blocks}
            sizeIncrement,            {in blocks}
            maximumSize: integer);    {in blocks}
```

where seg is assigned the segment number that has been created. There are two ways to use a segment once created. The first is simply to create it with a fixed size and use the entire segment at once. For example, when reading an entire file into memory. Use MakePtr(seg, offset, TypeOfPointer) to create a pointer of type TypeOfPointer in that segment at word offset "offset". The second way to allocate out of a segment is to use the standard Pascal NEW. NEW has been extended to have two forms. The standard form, NEW(p), allocates the pointer out of the default segment. The extended form, NEW(seg, allignment, p), allocates the storage out of the specified segment. Some buffers need to be specially aligned. For example, RasterOp buffers need to be on a multiple of 4. Do not use 0 for the alignment. For DISPOSE, only the pointer should be specified. NEW is implemented by a call to the procedure NewP in Dynamic. The user can call this procedure directly if he wants to specify the size of storage to allocate. NewP is defined as

```
Procedure NewP(seg: integer;
               allignment: integer;
               var p: MMPointer;
               size: integer);
```

The segment number of 0 is always defined to be the default segment for NewP and NEW. All other segment numbers should come from a prior CreateSegment. To calculate the size of a record or array, WordSize is a useful intrinsic. It returns the size of any PASCAL variable or type and can be used in constant or variable expressions. The user must remember the size used with NewP since DisposeP takes the size as a parameter.

```
Procedure DisposeP(var p: MMPointer; size: integer);
```

The size MUST be the same size used with NewP. One way to insure this is to store the size as a field in a record. As an example of NewP, we make a variable length array of strings:

```
Type
  s25 = String[25];
  NameDesc = RECORD
               numCommands: integer;
               recSize: integer;
               commands: array[1..1] of s25; {vbl length array}
             END;
  pNameDesc = ^NameDesc;
```

To allocate a pNameDesc with NUM names in the segment seg, the following would be done:

```
    var p: MMPointer;
        size: integer;
        names: pNameDesc;
    begin
    size := 2*WordSize(integer) + { for the 2 integers  }
            NUM*WordSize(s25);    { the variable part }
    NewP(seg, 1, p.p, size);
    names := RECAST(p.p, pNameDesc);
    names^.recSize := size;
    names^.numCommands := NUM;
{$R-}  {turn range checking off to assign names}
    for i := 1 to NUM do
       names^.commands[i] := '<some string>';
{$R=}  {return range checking to the previous state}
    end;
```

Since Dynamic uses special places in the segment to store the free list information used by NEW, it is bad practice to mix NEW and MakePtr on the same segment.

When a program requires a large amount of data, consider the swapping characteristics of the operating system. Since POS swaps an entire segment at once, a big segment will take much longer to read in and write out. Also, there may simply not be enough memory to hold the large segment and all other necessary data. Therefore, the user should divide the data into separate segments, each of which is about 10 blocks large. For example, this is what the editor does to hold the piece table.

3. Reading in Large Files.

There are a number of ways to read in a font or a picture from the disk. The fastest and most straightforward way is to use MultiRead. This is a special procedure that uses the micro-code's ability to read multiple blocks at once. The read, therefore, occurs at the maximum possible speed (the actual speed depends on how contiguous the blocks are on the disk).

To use multi-read on a file called FileName do the following:

```
var fid: FileID;  {imported from FileSystem}
    blocks, bits: integer;
    seg: Integer;
begin
fid := FSLookUp(FileName, blocks, bits);
if fid = 0 then {file not found}
else begin
     CreateSegment(seg, blocks, 1, blocks);  {allocate}
     MultiRead(fid, MakePtr(seg, 0, pDirBlk), 0, blocks);
     end;
end;
```

MultiRead takes a fileID, a pointer to the start of the block of memory, the first block to read of the file to read, and the number of blocks. The above code reads in the entire file.

If you do not wish to import MultiRead, you can read in each block of the file using FSBlkRead. Replace the MultiRead call above with the following

```
for i := 0 to blocks - 1 do
   FSBlkRead(fid, i, MakePtr(seg, i*256, pDirBlk));
```

The MakePtr creates a pointer to the i-th block (the i*256-th word) of the segment.

WARNING: The multi-block read exported by FileAccess does not work.

4. RasterOp and Line.

RasterOp and Line are the chief graphics primitives of the PERQ. Each is fast. TThe primitives allow drawing of rectangles and lines, respectively. RasterOp is described in the PERQ Pascal Extensions manual and Line is exported by the Screen module.

Use RasterOp to clear a rectangle (either white or black); transfer a picture from one place to another; or combine two pictures. Use Line to draw a single width line on the screen at any orientation.

RasterOp is a general utility. It can be used on buffers that are not on the screen. Therefore, it takes parameters that describe the dimensions of the buffer. For the Screen, the two constants SScreenW and SScreenP are exported by the Screen module. As a first example, we will clear an area of the screen 100 bits wide, 200 bits tall, starting at position (300, 400):

```
RasterOp(RXor, 100, 200, 300, 400, SScreenW, SScreenP,
                         300, 400, SScreenW, SScreenP);
```

We do this by Xoring the area with itself. Similarly, to clear an area to black, use the function RXNor. The function names are exported by the module Raster. To move a rectange from one area of the screen to another, simply use a different source and destination position. Remember that the destination is specified first.

To move a rectangle one bit down:

```
RasterOp(RRpl, 100, 200, 300, 400, SScreenW, SScreenP,
                         300, 399, SScreenW, SScreenP);
```

The position (0,0) is in the upper left corner; the lower right corner is (767, 1023). RasterOp does not validate the widths or positions so be careful. Be especially careful to avoid negative widths and heights since these are taken as large positive numbers. The available RasterOp functions are:

```
RRpl   {dest get src}
RNot   {dest get invert of src}
RAnd   {dest gets dest AND src}
RNAnd  {dest gets dest AND invert of src}
ROr    {dest gets dest OR src}
RNor   {dest gets dest OR invert of src}
RXor   {dest gets dest XOR src}
RXNor  {dest gets dest XOR invert of src}
```

RasterOp can also move a picture from or to an off-screen buffer. Suppose a picture is 543 bits wide and 632 bits high. The buffers used by RasterOp must be a multiple of 4 words in width. Therefore, allocate a buffer that is 36 words (=576 bits) wide and 632 bits high. This is 22752 words. Since segments can only be allocated on block boundaries, round up to 22784 words or 89 blocks and create a segment of this size and a RasterPtr to its start:

```
CreateSegment(seg, 89, 1, 89);
p := MakePtr(seg, 0, RasterPtr);
```

Now we might read a file into this buffer as described in Section 3. Next, we want to transfer the picture onto the screen, say at position (10, 100). We use

```
RasterOp(RRpl, 543, 632, 10, 100, SScreenW, SScreenP,
                          0,  0,   36, p);
```

The destination (given first) is (10, 100) on the screen, but the source is now the buffer. The bit width to transfer is 543 (the second argument), but the word width of the buffer is 36. (SScreenW is 48, the number of words across the screen). p is the pointer to the buffer. A picture can be transfered from the screen into a buffer, or between buffers in a similar manner.

If you want to allocate a buffer using NEW or NewP for RasterOping to or from, be sure to make the allignment 4.

Line is used for drawing straight, single width lines on the screen. It takes a source and destination x and y position, a style and a pointer to the buffer to draw in. Currently, it can only draw lines in buffers that have width 48 (e.g. the screen). Line is defined as:

```
Line(style: LineStyle; x1, y1, x2, y2: integer;  p: RasterPtr);
```

where the style is DrawLine, XOrLine or EraseLine. Use SScreenP for p.

5. Windows.

POS currently supports multiple, overlapping windows. However, POS does not know when two windows overlap. Thus all windows are "transparent" in that anything written to a covered window will "show through" any windows that are on top. Even with this restriction, windows are useful for a number of applications. For example, if multiple things are going on and the user wants to separate the input and output of each. The Screen package handles scrolling of the text inside windows automatically. Therefore separate windows scroll separately (if they do not overlap). This is useful, for example, in a graphics package where there are commands typed in a small window with the rest of the area used for the graphics (an example is the CursDesign program from the User Library).

The user must maintain the allocation of windows; the user tells the screen package where each window is and is expected to remember the number for each window. Window zero is reserved for the system and its size should not be changed. Use CreateWindow to create a new window. The parameters passed are for the outside of the window. There are two bits of border, then a hair line, then two more bits on each side. On the top there may be a title line which is a band of black with white letters in it. Once a window is created, it cannot be moved or re-signed.

Creating a new window automatically changes output to go to the new window. Given a set of windows, you can change amongst them by using the ChangeWindow command. The procedure GetWindowParms returns parameters of the current window. Unfortuneately, you must do transformations on the numbers returned to get the inside and outside areas of windows:

```
GetWindowParms(var windx: WinRange; orgX, orgY, width, height:
integer;
                    var hasTitle: boolean);
```

windx is the current window number and hasTitle tells whether there is a title line. Calculate the outside of the window as follows:

```
begin
orgX := orgX - 2;
width := width + 5;
orgY := orgY - 2;
height := height + 5;
if hasTitle then
   begin
   orgY := orgY - 15;
   height := height + 15;
   end;
end;
```

Calculate the inside of the window as follows:

```
    begin
    orgX := orgX + 2;
    width := width - 4;
    if hasTitle then
       begin
       orgY := orgY + 2;
       height := height - 4;
       end;
    end;
```

One thing to note: the title line for a window is written in the font that was in effect before the window was created. This font will also be remembered as the font to use the next time the window that was in use before the CreateWindow is used again.

6. Fonts.

The definition of fonts is given in the Screen module. Fonts currently can be variable width, but there is no kerning (the font must fit within the character block). A font starts with some global information: the height of the font in bits and the offset of the baseLine. Next is an array, which for each character has the position and width of that character in the font. A width of zero means the character is not defined. After this array are the actual bit pictures for the characters which are defined. Fonts can be created by using the FontEd program from the User Library available from the Sales department.

To use a font, it must first be loaded into memory. See the section on reading files above. The Screen package allows you to change the font to one you have defined. First, you should define a new window so that you don't change the font for the default system. Now simply call the function SetFont passing it a pointer to the top of the segment into which you read the font. If you wish to RasterOp a character (ch) using font FontP onto the screen by hand (at position (xPos, yPos)), use the following form (copied from SPutChr in Screen):

```
var Trik: Record Case Boolean of
                  true: (F: FontPtr);
                  false: (seg, ofst: integer);
                end;
begin
with FontP^.Index[ord(ch)] do
  if width > 0 then
      begin
      Trik.f := FontP;
      RasterOp(RRpl, width, FontP^.height, xPos,
                    yPos-FontP^.Base, SScreenW, SScreenP,
                    Offset, Line*FontP^.height, SScreenW,
                    MakePtr(Trik.seg, Trik.Ofst+#404, FontPtr));
      end;
 end;
```

The #404 is the size of the introductory part of a font. Trik is used to create a pointer to the actual bit pattern part of a font.

7. Cursors.

Unfortuneately, the term "Cursor" is used in two ways in PERQ-land. First, it is the position where the next character will be placed on the screen. This "cursor" is usually signified by an underline "_". The second "cursor" is the arrow or other picture that usually follows the pen or puck on the tablet. This section discusses the latter form.

The picture in the cursor can be set by the user. POS currently uses a number of different pictures. The default arrow cursor, the "scroll" and "do-it" cursors for PopUp menus, the hand that moves down the side of the screen, and the Busy Bee are all examples of cursors. The program CursDesign from the User Library can be used to create cursors. Once a picture has been created, it can be read into Memory from the file (see above) and then copied into the Cursor. Each cursor is 56 bits wide and 64 bits tall which comes to 4 words wide and 64 bits tall or exactly one block. Therefore a file with one cursor in it can be read in directly into the cursor buffer. The definition of the cursor and all utility procedures for manipulating it are in IO_Others.

```
var curs: CurPatPtr;
begin
New(0,4,curs);
Fid := FSLookup(CursorFile, blks, bits);
FSBlkRead(fid, 0, RECAST(curs, pDirBlk));
end;
```

Note that the cursor buffer must be quad-word aligned (since a RasterOp is done from it by the system). To set a cursor, use the function IOLoadCursor. It takes a CurPatPtr and two integers to tell it the x and y offsets in the cursor from where the cursor is positioned. Thus, for a "bull's eye" cursor where the center is the interesting point, the offsets would be the offsets from the top left of the center. For a right pointing arrow, the offsets would describe the point of the arrow. The user then does not need to compensate when reading the cursor position. IO_Others exports the cursor DefaultCursor which is the upper-left pointing arrow.

The cursor can be used in a number of ways. If you want the cursor to follow the tablet and then read the tablet coordinates, use the cursor mode TrackCursor.

```
IOCursorMode(TrackCursor);
```

Be sure to turn the tablet on using IOSetModeTablet(relCursor). If you want to explicitly set the position of the cursor, use cursor mode IndepCursor. To set the cursor position, use the function IOSetCursorPos(x,y); Note that if you set the cursor position in Track mode, it will be overwritten almost immediately by the position of the tablet. You can still read the tablet in IndepCursor mode if it has been turned on; the tablet position is simply not used to set the cursor position.

To read the tablet position, use the function IOReadTablet. It returns the last x and y position read from the tablet. If the pen or puck is away from the tablet, it may be an old point. It is not possible to tell if the tablet is sensing the pen or puck. The buttons can be read using the variables TabSwitch, TabYellow, TabBlue, TabWhite, and TabGreen. TabSwitch tells if any button was pressed. For a puck, the other booleans tell which button it was. For a pen, the "colored" booleans will always be false. These booleans are true while the button is held down. The user is required to wait for a press-let up event:

```
repeat until tabswitch;
while tabswitch do;
{ read tablet position, or whatever }
```

The Cursor functions determine how the cursor interacts with the picture on the screen under the cursor. The cursor function also determines the background color. The even functions have zeroes in memory represented as white and ones as black (this is the default: white background with black characters). Odd functions have zeroes represented as black and ones as white. The functions are as follows (inverted means screen interpretation as just described):

CTWhite:	Screen picture is not shown, only cursor.
CTCursorOnly:	Same as CTWhite only inverted.
CTBlackHole:	This function doesn't work.
CTInvBlackHole:	This function doesn't work either.
CTNormal:	Ones in the cursor are black, zeros allow screen to show through.
CTInvert:	Same as CTNormal only inverted.
CTCursCompl:	Ones in the cursor are XORed with screen, zeros allow screen to show through.
CTInvCursCompl:	Same as CTCursCompl only inverted.

8. Reading Characters from the Keyboard.

The normal PASCAL character Read waits for an entire line to be typed before returning any characters. This allows editing of the line (backspace, etc.) as described in the PERQ Introductory Manual. If you want to get the characters exactly when they are hit, you must call IOCRead in IO_Unit. The normal form for this call is

```
If IOCRead(TransKey, c) = IOEIOC then { c is a valid character }
```

where IOEIOC is a constant defined in the module IOErrors and c is a character variable. If IOCRead returns some value other than IOEIOC, then no character has been hit. "Transkey" tells IO that you want the standard ASCII interpretation of the character. If you use "KeyBoard" instead, you will get the actual 8 bits returned by the keyboard. This code allows you to distinguish the special keys (INS, DEL, HELP, etc.) from the other keys and allows you to distinguish CONTROL-SHIFT-key from CONTROL-key. You will have to experiment to get the code for the desired key. There is no way to tell when a key has been let up.

IOCRead does not write out the character typed. If you want it printed, you should use Write(c). If you want to print all the special symbols in the font file (there is a picture associated with every control character), you can set the high bit of the character. This prevents the Screen package from interpreting the character as its special meaning if any. Thus, you could print the picture for RETURN by using

```
Write(chr(LOr(RETURN, #200)));
```

IOCRead also does not turn on the input marker ("_") which shows the user that he is supposed to type something. Do a SCurOn (from Screen) before requesting input and an SCurOff when done to make the underline prompt appear.

The HELP key and ^C are handled specially by the IO system. If the HELP key is hit, an exception is raised. If you do not handle this exception (called HelpKey, exported by System), "/HELP<CR>" will be put into the input stream as if typed. If you do handle this exception, you can put chr(7) into the input stream: the code for HELP. When ^C is typed, the exception CtlC is raised (also defined in System). If not caught, nothing special is done until the second ^C is hit when CtlCAbort is raised. This causes the program to exit. Note that the ^C's are put into the input stream. ^SHIFT-C causes a separate exception to be raised. If the user wants one ^C to do something special in a program (for example, abort type-out and go to top level as in FLOPPY), put the following Handler at the top level:

```
Handler CtlC;
   begin
   WriteLn('^c');
   IOKeyClear;                {remove the ^C from input stream}
```

```
        CtrlCPending := false;    {so next ^C won't abort program}
        goto 1;                   {top of command loop}
        end;
```

(IOKeyClear comes from IO_Others.) Another special character to know about is ^S. This character prevents any further output to the screen until a ^Q is typed. If you want to disable this processing, simply set CtrlSPending to false after every character is read.

9. CmdParse and PopCmdParse.

CmdParse and PopCmdParse export a number of procedures that help read and parse strings of commands and arguments. Procedures exist for handling command files (which may be nested), for parsing a string containing inputs, outputs and switches into its components, and for getting a command index from a string or a PopUp menu.

The modules CmdParse and PopCmdParse document how each of the procedures work. This section provides an example of how to use the parsing procedures in CmdParse.

```
var ins, outs: pArgRec;
    switches: pSwitchRec;
    switchAr: CmdArray;
    err: String;
    ok, leave: boolean;
    c: Char;
    s: CString;
    isSwitch: boolean;
    i: integer;
  begin

  <assign all switches to SwitchAr>

  c := NextString(s, isSwitch);  {remove "<utility>"}
  if (c<>' ') and (c<>CCR) then
      StdError(ErIllCharAfter, '<utility>', true);
  ok := ParseCmdLine(ins, outs, switches, err);
  repeat
     if not ok then StdError(ErAnyError, err, true);
     while switches <> NIL do {handle all the switches}
        begin
        ConvUpper(switches^.switch);
        i := UniqueCmdIndex(switches^.switch,
                 switchAr, NumSwitches);
        case i of
             1 : <handle switch # 1>
             2 : <handle switch # 2, etc.>
             otherwise: StdError(ErBadSwitch,
                              switches^.switch, true);
             end;
        switches := switches^.next;
        end;
     if (outs^.name <> '') or (outs^.next <> NIL) then
         StdError(ErNoOutFile, '<utility>', true);
     if ins^.next <> NIL then
         StdError(ErOneInput, '<utility>', true);
     if ins^.name = '' then
        begin
        Write('<Prompt for argument>: ');
        ReadLn(s);
        ok := ParseStringLine(s, ins, outs, switches, err);
        leave := false;
        end
```

```
    else begin
         leave := true;
         if not RemoveQuotes(ins^.name) then
             StdError(ErBadQuote, '', true);
         FSRemoveDots(ins^.name);

         <handle the argument>

         end;
until leave;
end;
```

PERQ File System Utilities Manual

Brad A. Myers

This manual describes the PERQ file system utilities: Partition, Scavenger, MakeBoot, and FixPart. Other utilities are described in the manual "PERQ Utility Programs Manual." In addition, relevant parts of the file system are described so the programs can be understood.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Table of Contents 04 Feb 82

1. Preface: Notation Conventions.

The notations used below have been clearly and consistently used throughout this document.

SYMBOL	MEANING
[]	optional feature
{}	0 to n repetitions
CAPITALS	literal
lowercase or LowerCase	metaname
^	control key
CR	carriage return
SHIFT	shift key
|	or

2. Introduction.

The PERQ has a hierarchical, multi-directory file system which supports search lists and noncontiguous files. Devices (e.g., hard and floppy disks) are divided into a number of sections called "partitions." Each partition can contain any number of directories. The main directory in a partition (the one you get if you simply specify the partition name) is the Root directory for that partition. All directories can contain other directories and files. Directories are stored as standard files and can be accessed by any program. However, the system knows special things about their format. All files in this file system can be noncontiguous; blocks for files may be scattered throughout the partition in which they reside. The naming conventions for filenames are described in detail in "The PERQ Introductory User Manual" but all are of the form:

```
[[[device]:partition]>]{directory>}[filename]
```

Files may be stored on floppy disks in two ways. The FLOPPY program can be used to transfer files to and from the floppy, or the floppy can be initialized as part of the file system. These types of floppy disks are called "FLOPPY floppies" and "filesystem floppies" respectively.

NOTE: Floppies of one type CANNOT be used for the other; the formats are totally incompatible. Floppy disks that contain files transferred by the program FLOPPY must be accessed by the FLOPPY program (and not by the file system directly), and FLOPPY cannot read a file system floppy.

To create a file system floppy, use the Partition program described below.

The user programs that handle the file system are described in the "PERQ Utility Programs Manual" and only Partition, Scavenger, MakeBoot, and FixPart are described in detail here.

To create a file system on a blank disk, it is necessary to run the command file "InitDisk.Cmd" which is found on the boot floppy in every release package. A heavily-commented printed copy of this command file is provided as instructions for how to initialize a blank device. To create a file system on a floppy, use the Partition program described below. The manual "How To Make a New System" describes the process necessary to create a system using the MakeBoot program. This manual provides the background necessary to understand these processes.

3. Partitions and the Partition Program.

The partitions on a device are restricted to be fewer than 32768 blocks, where each block is 256 words (512 bytes, 4096 bits). The block size is fixed for all devices. On a 12-megabyte disk or a floppy, the entire device can be in one partition. On a 24-megabyte disk, at least 2 partitions are needed. We recommend, however, that each partition have 10080 or fewer blocks in them, otherwise Scavenger (described below) cannot handle the partition in one pass. All partitions must start and end on cylinder boundaries; the Partition program enforces this constraint.

No file or directory can be in more than one partition (a file cannot cross a partition boundary). Therefore, having free blocks in one partition does not prevent you from running out of free blocks in another.

The Partition program is used to create and modify the partitions on a device. Creating a partition usually destroys all old data in the area where the partition is made. The device should first be formatted (by using DTST on hard disks or FLOPPY for floppy disks). After formatting, Partition is the first program to run.

Partition asks whether you want to "debug". If you answer "YES", then nothing will be modified on the device. Answer "NO" if you want to make modifications.

Next, Partition requires that you tell it what device you want to modify. The choices are the harddisk or floppy. If you specify the harddisk, Partition checks to see whether the disk is a 12 or 24-megabyte disk. It then asks for confirmation of the size chosen. If you specified the floppy, you then have to tell partition whether the floppy is or single or double-sided. A floppy that is formatted on both sides can be partitioned as either single or double sided.

Next, the program asks if you want to partition the entire disk. Answer "YES" if you are starting from scratch (for example, on a newly formatted floppy or when installing the file system on a new machine). If you want to modify an existing device, answer "NO".

If you answered "YES" to initialize the entire disk, then the program asks for information about each partition in turn. The device must first be given a name (eight or fewer characters). Next, each of the partitions must be given a name (also eight or fewer characters). Examples of partition names used for the hard disk include "boot", "user", and "exp". There is no problem with having a partition with the same name as a device, but all the partitions must have unique names. The size of each partition is also requested. As mentioned earlier, we have found that 10080 or fewer blocks are desirable; otherwise Scavenger (described below) cannot handle the partition in one pass. This means there are 3 partitions on a 12-megabyte disk and 5 on a 24-megabyte disk. The minimum size for partitions is 120 blocks on a 12 megabyte disk, 240 blocks on a 24-megabyte disk, and 6 blocks on a floppy disk. The sizes of all partitions must be multiples of the mimimum size for

that device.

For each partition on the device, the program asks whether to initialize the partition pages. This process puts all the pages in the partition on the free list and therefore should be done for a new device. The next question is whether to test after initializing the pages. Although this slows down the initialization somewhat, it is good practice to do this testing. If any pages are found to be bad during testing, they are removed from the free list so they will never be accessed again. The final question asked before initialization is whether to write every page twice. This provides some additional protection from bad pages since random data is written into the header and body of each block. It has been found in practice that some bad blocks on the disk will pass the first test and fail this one.

After all the blocks on the device have been included in a partition, the program displays some data about the device and asks whether to remount the device. Only mounted devices can be accessed, so if you plan to use the device, say "YES".

If you said that you did not want to initialize the entire device, the program reads the device information block (which is in a reserved, fixed location on cylinder 0) to find what partitions are currently on the device. These are displayed in the order they appear on the disk. The first question asked is whether to rename the device.

NOTE: Before renaming a device or partition it is important to note that every Run file on the device has incorporated in it the device and partition name where it was linked. If you rename the device, then no program on that device can be run (including the Shell, Login, Partition, for example). If you rename a partition, no program in that partition can be run.

If you did not rename the device, you are asked which partition you want to modify. Type the name of one of the partitions. The program then asks what you want to do to the partition. You can split a partition into two parts, merge a partition with another, initialize a partition, or change the name of a partition.

It is never a good idea to split a partition with files in it since files cannot cross partition boundaries. Any file which has blocks in both partitions will be destroyed. It is safe to split an empty partition or one that you plan to erase. The program asks how much of the partition to leave with the old name; the rest of the blocks in the partition will be put in the partition just created. The program will ask if you want to initialize the partition pages to create a new free list. This is recommended. Next, a name for the new partition is solicited and the pages are initialized if desired.

It is much safer to merge two partitions together than to split one apart. You specify the first partition and it is joined with the partition which is next on the disk. This is the only time the

order of the partitions on the disk matters. If you want to erase the new, bigger partition, say that you want to initialize the pages. If not, then all the files on both partitions can be saved. Just after Partition exits, you must run Scavenger program (see below) on the new partition. When running the Scavenger in this case, be sure to tell it to rebuild the directories so that the directories of the two partitions can be joined together.

Partition also can change the name of a partition. Do not change the name of the partition that is used in the current path since the default path name then becomes invalid. In addition, all entries in the search list that refer to the partition renamed will no longer work. After a rename, therefore, the system may not be able to find the Shell or any other programs.

Finally, you can initialize the partition. This is a fast way to delete all the files in the partition. After asking whether to initialize the partition, the program asks whether to initialize the partition pages. If you answer "NO", you must use Scavenger program to recreate the directory immediately after running Partition. There are few reasons to initialize the partition without initializing the pages.

The Partition program can take a switch on the command line. If it is invoked with the /BUILD switch, then the entire device is partitioned. Note that the arguments must be specified on the command line. This is a dangerous thing to do and it is only recommended for command files which bring up an entire disk. The format for this switch is

Partition/Build <dev> <deviceName> <Partname> <PartName> ...

where <dev> is either "H" for the hard disk or "F" for the floppy. The device name is given next followed by enough partition names for the entire device. The partition names are used in the order specified so the first name will be used for the first partition on the disk. All partitions except the last will be the standard size (10080 blocks). If extra partition names are specified, they are ignored. If the device seems to be already formatted, the program requires confirmation before erasing the device. If the user answers NO, then partition is run the normal way asking the user all the questions.

4. The Scavenger Program.

The Scavenger program fixes a partition on a device that contains useful files. It checks all files for consistency, rebuilds the free list, and creates a new directory structure for the partition. The Scavenger should only be run on devices that have already been initialized by the Partition program. The Scavenger checks and fixes some system information and then checks all files in a partition for consistency and recreates the free list. It can also recreate the directories. The Scavenger also removes bad boots. The Scavenger should be run whenever an inconsistency is found in the file system or when some program aborts and asks you to run the Scavenger.

WARNINGS: 1) As with the Partition program, do not type Control-C (^C) to Scavenger after it has begun writing on the device. During the read pass, it is safe to ^C. If you type ^C after it begins rebuilding the directory, you may not be able to access anything in the directory. If this happens, rerun Scavenger from another partition.

2) Scavenger will not be able to recreate the directory if there are no free blocks in the partition. Therefore, if your partition is full, you must delete some files before running Scavenger. If you cannot delete any files due to a bad directory and there are no free blocks, then there is currently no way to rebuild the directory. In this case, you must initialize the partition, thus losing all files there. Fortunately, we have never seen this happen in practice.

Scavenger program fixes one partition at a time. It is possible (and sometimes necessary) to scavenge partitions other that the one you are currently running in. It is usually safe, however, to scavenge the current partition.

Scavenger program has three separate phases. In phase one, it checks and updates some of the system information. In this phase, bad boots are deleted. If a boot has been defined by MakeBoot (see below) but either the microcode or system code files have been deleted, the boot is known to be bad. The Scavenger cannot tell, however, if a boot file is deleted and another created in the same place before Scavenger is run. In this case, the boot will seem valid but will not work.

During the second phase of the scavenge, the partition specified by the user is checked for consistency. All blocks are read and a new free list is generated in ascending disk order (the old free list is discarded). In addition, blocks that are not readable are marked as "bad", and if they cannot be rewritten, they are marked as "incorrigible" and removed from the file system. All blocks that

were in files containing bad or incorrigible blocks are put in the bad file. In addition, any malformed chains are added to the bad file. The user is asked for a name for this bad file in phase three of Scavenger.

In phase three, the directories for the partition are completely rebuilt. Scavenger will delete the old directories, if desired. Otherwise, the old directories are marked as such and their names have a "$" added to the end. If there is not enough room for copies of all the directories in the partition to be created, Scavenger will crash leaving the directories only partially created. In this case, you have to delete some of the files in the directory and then rerun Scavenger.

Before entering any name in a directory or creating a new directory, Scavenger first checks to make sure the name seems reasonable as a filename. Certain characters are not allowed in filenames. These include any control characters, "<", "/", ":", and " ". In addition, the name may not end with a ">" or contain ">..>" or ">.>". If a bad name is found, or two files have the same name, Scavenger requests a new filename from the user. After Scavenger is finished, you can examine the files with bad names to see whether they contains any useful information. If so, rename or edit the files to recover the data. Otherwise, simply delete the files.

The Scavenger in this pass also makes sure the length of all files are correct and allow you to specify a new length. Note that this refers to the stored length rather than the number of blocks in a file. Certain files, like directories and swap files, do not bother to set the length field. File lengths usually become wrong when the file is opened and written but not closed properly. This, for example, happens when a transfer is aborted.

The Scavenger asks the user a number of questions before it begins processing the partition. First, it asks whether to look at the floppy or hard disk. It then checks that device to see how big it is and then asks if its choice is correct.

When it has this information, Scavenger asks if it can make changes to the device in the first two phases of the program (the directory fixing is handled later). This is like the "debug option" for the Partition program. If you answer "NO", then Scavenger checks the partition for errors and reports them but does not fix anything. If you are running Scavenger only to fix the directory, it is about twice as fast to answer "NO" to this question; otherwise, "YES" is a good idea.

Next, Scavenger asks if it should do logical block number consistency testing and serial number consistency testing. The header of every block contains information about the state of that block. The information includes the count of the block in the file (is it the first, second, etc.) and a two-word identifier for the file the block belongs to. These numbers are checked for correctness if you answer "YES". If the Scavenger continually aborts due to FullMemory, answer "NO" to one of these questions. To avoid the FullMemory condition on machines with 256k bytes of main memory, the default

for serial number checking is no. Just type CR to get the default answer.

Then Scavenger asks if there is enough memory to do the scavenge in one pass. If your partition has 10080 or fewer blocks in it and if the screen has been shrunk (the Shell shrinks the screen when it knows you are running the Scavenger), then the answer is "YES". If your partition is bigger than 10080 blocks, then answer "NO". Three passes will then be used and the program will be correspondingly slower.

The next question is how many retries for a suspect read. The default is 15. If the Scavenger aborts with the error "block xx was found bad during... but was thought to be good before" or if it aborts during rebuilding of the directories, try rerunning the Scavenger with a smaller number. One is the smallest valid answer.

If you answered "YES" when asked if Scavenger could change your disk, you will be asked three more questions about ways Scavenger might change the disk. First, Scavenger asks whether it should delete temporary files. Temporary files exist for swapping; all user files are permanent. Second, the Scavenger asks if it should delete old bad segments. As described above, files with bad blocks in them are marked as bad. If you answer "YES" to this question, then the bad file created by the previous scavenge of this partition is added to the free list. Finally, Scavenger asks if it can rewrite bad blocks. If a block cannot be successfully read in the specified number of retries, it is possible that writing new data onto the block will fix the problem. However, for our hard disks, this seems to have a small chance of fixing the problem. Therefore, the default answer is "NO". However, if you answer "YES", the bad blocks will be rewritten. If the write or a subsequent read fails, then the block is "incorrigible", otherwise it is "bad". If rewritting is not permitted, then the block is marked "incorrigible" as soon as the read fails. If there are only transient read errors, the block is left alone.

After you answer all of the questions, Scavenger can get to work. The title line of the window is updated to show what Scavenger is working on. In addition, various cursors are used to show the progress of the different passes. First, Scavenger does phase one checking as described above, fixing any discrepancies if you allowed changes. If Scavenger finds a problem with the partition or device information blocks it cannot fix, it asks for help. If you cannot figure it out either, the problem may be that: 1) you specified the wrong device type to the first question; 2) the device is not a file system device (e.g. a FLOPPY floppy); 3) the device has not been initialized; or 4) the device has been messed up beyond repair. Unfortunately, the only fix in this case is to re-partition the device from scratch.

After the device and partition information checks out, Scavenger displays a list of the partition names and asks which one it should work on. Type the name of the partition to be scavenged. Next, Scavenger will display some information about the specified partition. The values are those stored in the partition information

block before the scavenge. Now Scavenger makes a read pass through the partition building tables of each block's next and previous link (this is the data usually visible in the lower portion of the screen). The pass is done in eight parts for efficiency. Therefore, the Scavenger cursor goes down the screen eight times before the next step. The tables built by Scavenger are now checked for consistency, and the cursor changes to show that checking is in progress. If any loops are found, Scavenger breaks the loops and blinks the screen to show that a loop has been fixed. Afterwards, if changing the device is allowed, the Scavenger rebuilds the free list. This requires a write pass for all blocks on the free list and a write cursor is displayed. If any bad blocks were found, some more reads and writes are necessary to make the bad blocks into a well-formed chain. After this pass, Scavenger writes the new partition information block.

Next, the directory building pass is started. Scavenger asks whether it should rebuild the directory. Sometimes it recommends that you do this, otherwise there is no default. If you suspect there is a problem with the directories and if there are enough free blocks, answer "YES". The old directories will be deleted, if you so specify. Otherwise they are saved for later inspection. A "$" is appended to the end of their names and their file type is changed to ExDirFile (directories all have the type DirFile). New directories are then created whenever needed. This means that empty directories are not recreated. The old directories are just files that you can delete after the scavenge.

Note: This scheme makes it easy to recover from overwriting or deleting a directory since the directory reappears after a scavenge.

As described above, the Scavenger checks and allows fixing file lengths if desired. For each file, it checks the stored length with the actual number of blocks in the file. If they do not match, it allows you to specify a new stored length. This can be any value, but making it bigger than the number of blocks in the file is not recommended. The default for the stored length is the number of blocks in the file. The Scavenger does not check the lengths for directory files or files with their type field set to "SWAPFILE."

Each file has a table which points to each logical block of the file. This allows the file system to find a random logical block without searching down the chain from the file start. This table is called the "Random Index Table." Scavenger, as part of the directory building phase, can rebuild the random indices for all files. There is a separate question for this with a default answer of "NO". There is usually no reason to rebuild the indices unless Scavenger asks you to. Building the random indices for large files takes a long time.

If a bad file was created, Scavenger will ask for a name for the that file at the end of the directory building phase. If you allow Scavenger to enter and fix the indices for the bad file, you can then type and edit it as a normal file. In this way some useful

information may be reclaimed.

5. MakeBoot.

The MakeBoot program creates new systems. An overview of its use appears in the manual "How To Make a New System". The "PERQ Introductory User Manual" describes the booting process. MakeBoot creates a boot file taking a stand alone run file (such as a system) and then associates this boot file with a letter. The lower case letters are assigned to the hard disk and the upper case letters are assigned to the floppy disk. The default that is used when no keys are held down is lower case "a". Boot letters can be freed of the associated boot by deleting the system and/or interpreter boot files.

Any program can be made to work "stand-alone" (so it can be booted) by initializing various modules as the System program does. It is generally not necessary or desirable to have programs other than the System be stand-alone.

The run file name given to MakeBoot determines on which device and partition the boot will be. MakeBoot takes the directory part of the file name and uses that to determine on which device and partition to put the boot. Therefore, to make a boot somewhere, first copy the run file to that partition. After specifying the run file, Makeboot asks for the configuration file. This file tells MakeBoot which System modules are swappable. The format of the file is:

Module-name swappability

where swappability is "sw" for swappable, "um" for unmovable (stronger than unswappable), or "us" or blank for unswappable. The default is unswappable so only the modules in your system that you want to be swappable or unmovable need to be listed. The default system config file (named System.nn.Config) is:

```
*SAT* UM
*SIT* US
*Cursor* UM
*Screen* UM
*Font* US
*IO* UM
System SW
Stream SW
Writer SW
IOErrMessages SW
Loader SW
Reader SW
Perq_String SW
Screen SW
FileSystem SW
Code SW
GetTimeStamp SW
FileDefs SW
Memory SW
IO_Init SW
```

RunRead SW
FileDir SW
Scrounge SW

The system data segments that the hardware uses are required to be unmovable, the data used by software (*SIT* and *FONT*) are required to be unswappable, and everything else that is not used by the swapping system itself can be swappable.

The next question MakeBoot asks is whether to write the boot microcode onto the device. No matter how many boot letters are defined for a device, there is only one set of boot microcode so this only needs to be written when putting the first boot onto a device. The standard boot microcode files are "SysB" and "Vfy".

There are two files associated with each boot letter. The system boot file is Pascal and the interpreter boot file is microcode. The system boot file is created by MakeBoot by reading the supplied run file. The standard microcode is usually used with all boot files. It is used by MakeBoot to create the interpreter boot file if you so specify. If you have already created a boot for the current letter and you have not changed microcode, it is not necessary to make a new interpreter boot file, but it never hurts to do so. If you want to load the standard microcode and it is found by MakeBoot, type CR when it asks for an interpreter microcode file.

Included in the system code is the default character set font. MakeBoot looks for the default font (currently, "Fix13.kst") in all the search paths. If it is not found, you will have to supply a font file name.

Note: For the Editor and certain other programs to work, the default font must be fixed width and thirteen bits high and nine bits wide.

MakeBoot puts the output boot files wherever you specify but it is important that the interpreter and system boot files be in the same partition. The device and partition in which the boot file is created will be the default path after the boot. This means that there must be at least a "Login.nn.run" and a "Shell.nn.run" (where "nn" is the version number of the system run file), in the root directory of the partition. It doesn't matter if the boot files are in a subdirectory in the partition; the run files mentioned above must be in the Root directory.

The MakeBoot program will take a switch on the command line. If it is invoked with the /BUILD switch, all arguments are specified on the command line and the user is not asked any questions. The format for this switch is:

MakeBoot [<dir>]System.<nn>/Build <bootKey>

where <dir> is an optional directory, <nn> is the system version number and <bootKey> is the character to boot from. Makeboot then uses the default answers for all questions.

6. FixPart.

FixPart is an experimental program for fixing the Device and Partition information blocks. It is not recommended that customers try to use it without assistance. Unlike the other programs described above, FixPart is only partially automatic and can cause a lot of damage. Unfortunately, it is currently the only way to fix bad partition and device information blocks.

Note: It is very rare that the device and partition information blocks get broken, so Scavenger should always be run first to see if the problem is actually elsewhere.

FixPart first asks if you are sure you want to run the program. Next, it asks for the device type. It then goes through and checks each partition for consistency with the other partitions and with the device information block. If a name is dubious it asks if it is valid or not. After all partitions are checked, the program gives a summary of the errors. If none were found, then it exits; otherwise, you can specify new start and end addresses for the partition and fix the names.

If the device information block is not writeable, then you have to reformat the entire device and, unfortunately, lose all the data on it. If one of the partition information blocks is not writeable, then you may be able to save some information by using the partition program to join the partition with the bad information block to the partition before and then scavenging. If it is the first partition, however, your device will have to be reformatted.

PERQ Micro-Programmer's Guide

Brian Rosen

John P. Strait

This manual provides an introduction to the PERQ micro-programmable CPU.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

The information in this document is subject to change without notice and should not be construed as a commitment by Three Rivers Computer Corporation. The Company assumes no responsibility for any errors that may appear in this document.

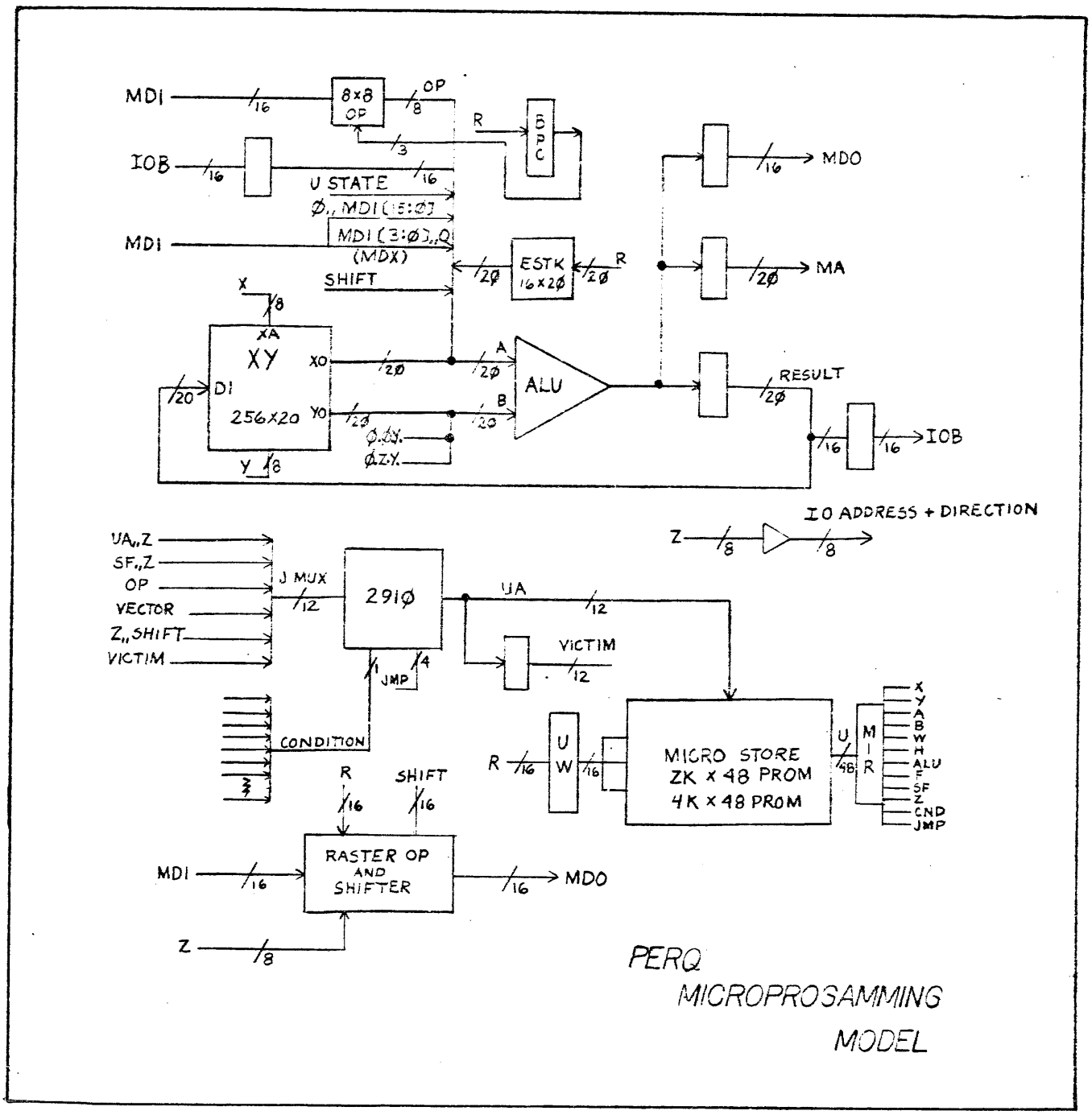
Three Rivers Computer Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distributed with this document to request users' critical evaluation to assist us in preparing future documentation.

Table of Contents

Introduction to the Hardware

PERQ is implemented with a high-speed microprogrammed processor which has a 170 nanoseconds microcycle time. The microinstruction is 48 bits wide. Most of the data paths in the micro engine are 20 bits wide. The data coming in and out of the processor (IO and Memory data for instance) are 16 bits wide. The extra 4 bits allow the microprogrammed processor to calculate real addresses in a 1 megaword addressing space. The assumption is that virtual addresses are kept in a doubleword in memory but calculations on addresses can be single precision within the processor. The programmer of the virtual machine never sees the 20 bit paths.

The major data paths are diagrammed below:



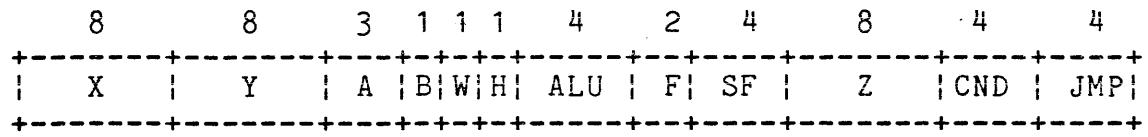
The XY registers (256 registers x 20 bits) form a double ported file of general-purpose registers. The X port outputs are multiplexed with several other sources (the AMUX) to form the A input to the ALU. The Y port outputs, multiplexed with an 8- or 16-bit constant via the BMUX, form the B input to the ALU. The ALU outputs (R) are fed back to the XY registers as well as the memory data output and memory address registers. Memory data coming from the memory is sent to the ALU via the AMUX. The IO bus (IOB) connects the CPU to IO devices. It consists of an 8-bit address (IOA) which is driven from a microword field and a 16-bit bidirectional data bus (IOD) which is read via AMUX and written from R.

Opcodes and operands that are part of the instruction byte stream are buffered in a special 8 x 8 RAM (the OP file). The OP file is loaded 16 bits at a time from the memory data inputs. The output of the OP file is 8 bits wide and can be read via AMUX or can be sent to the micro-addressing section for opcode dispatch. The read port of the OP file is addressed by the 3-bit BPC (Byte Program Counter).

A shift matrix (SHIFT), which is part of the special hardware provided for the RasterOp operator, can be accessed by loading an item to be shifted via the R bus, and reading the shifted result on AMUX.

A 16-level push down stack (ESTK) is written from R and read on AMUX. The stack is used by the Q-code interpreter to evaluate expressions. BPC and the microstate condition codes can be read as the Micro State Register (USTATE) via AMUX.

Micro Instruction Format



Field	Width	Use
X	8	Address for X port of XY. Also address used to write XY.
Y	8	Address for Y port of XY. Also low 8 bits of constant.
A	3	AMUX Select: 0 SHIFT 1 NextOp 2 IOD 3 MDI Memory Data Inputs, AMUX[19..16] := 0 AMUX[15..00] := MD[15..00] 4 MDX Memory Data Input extended AMUX[19..16] := MD[03..00] AMUX[15..00] := 0 5 USTATE 6 XY (RAM) 7 ESTK
B	1	BMUX select: 0 = XY[Y], 1 = Constant.
W	1	Write: XY[X] := R if W = 1.
H	1	Hold: If set, do not allow IO devices to access memory. Also used with JMP field to modify address inputs.
ALU	4	ALU function: 0 A 1 B 2 not A 3 not B 4 A and B 5 A and not B 6 A and B 7 A or B 10 A or not B 11 A nor B 12 A xor B 13 A xnor B 14 A + B 15 A + B + OldCarry 16 A - B 17 A - B - OldCarry

F 2 Function: Controls usage of SF and Z fields.

F	SF use	Z use
0	Special Func.	Constant/Short Jump
1	Memory Control	Short Jump
2	Special Func.	Shift Control
3	Long Jump	Long Jump

SF 4 Special Function: Upper 4 bits of address for long jump and memory control functions (see F). When used as Special Function:

0	Long Constant
1	ShiftOnR
2	StackReset
3	TOS := (R) (Top Of ESTK)
4	Push (ESTK)
5	Pop (ESTK)
6	CntlRasterOp := (Z)
7	SrcRasterOp := (R)
10	DstRasterOp := (R)
11	WidthRasterOp := (R)
12	LoadOp (OP := MDI)
13	BPC := (R)
14	WCS[15..00] := (R)
15	WCS[31..16] := (R)
16	WCS[47..32] := (R)
17	IOB Function

Z 8 Low 8 bits of Jump Address, high 8 bits of Constant, Shift Control (see F), or IOB address.

CND 4 Condition (what to test during conditional jump):

0	True - always jump
1	False - never jump
2	IntrPend - interrupts pending
3	unused
4	BPC[3] - OP file is empty
5	C19 - no carry out of bit 19 of the ALU
6	Odd - ALU bit 0
7	ByteSign - ALU bit 7
10	Neq - not equal to
11	Leq - less than or equal to
12	Lss - less than
13	OverFlow - 16 bit overflow in the ALU
14	Carry - carry out of bit 15 of the ALU
15	Eql - equal to
16	Gtr - greater than
17	Geq - greater than or equal to

JMP 4 Jump Control: See AMD 2910 documentation for further details.

CIA = Current Instruction Address.
NIA = Next Instruction Address.
Addr = SF,,Z (Long) or CIA,,Z (Short).
ZAddr = Z(UpperBits),,0,,Z(LowerBits).
S = internal address register.
CSTK = top of five level call stack.
Push = push CIA+1 onto call stack.
Pop = pop call stack.

Code	Name	Pass	Fail
0	JumpZero	NIA:=0	NIA:=0
1	Call	NIA:=Addr Push	NIA:=CIA+1
2	NextInst/ReviveVictim		
	H = 0	NIA:=OP +ZAddr	NIA:=OP +ZAddr
	H = 1	NIA:=Victim	NIA:=Victim
3	GoTo	NIA:=Addr	NIA:=CIA+1
4	PushLoad	NIA:=CIA+1 Push S:=Addr	NIA:=CIA+1 Push
5	CallS	NIA:=Addr Push	NIA:=S Push
6	Vector/Dispatch		
	H = 0	NIA:=Vector +ZAddr	NIA:=CIA+1
	H = 1	NIA:=Dispatch +ZAddr	NIA:=CIA+1
7	GotoS	NIA:=Addr	NIA:=S
10	RepeatLoop		
	if S <> 0	NIA:=CSTK S:=S-1	NIA:=CSTK S:=S-1
	if S = 0	NIA:=CIA+1 Pop	NIA:=CIA+1 Pop
11	Repeat		
	if S <> 0	NIA:=Addr S:=S-1	NIA:=Addr S:=S-1
	if S = 0	NIA:=CIA+1	NIA:=CIA+1
12	Return	NIA:=CSTK Pop	NIA:=CIA+1

13	JumpPop	NIA:=Addr Pop	NIA:=CIA+1
14	LoadS	NIA:=CIA+1 S:=Addr	NIA:=CIA+1 S:=Addr
15	Loop	NIA:=CSTK Pop	NIA:=CIA+1
16	Next	NIA:=CIA+1	NIA:=CIA+1
17	ThreeWayBranch		
	if S <> 0	NIA:=CIA+1 Pop S:=S-1	NIA:=CSTK S:=S-1
	if S = 0	NIA:=CIA+1 Pop	NIA:=Addr Pop

The Z field is used for many things: as part of a jump address, the upper 8 bits of a constant, Shift Control, and as an IOB address. The F field decodes do not necessarily enforce restrictions on the use of the Z field, they merely enable some of them. In particular, B = 1, SF = 0, and F = 0 or 3 selects a long constant using the Z field. When F <> 2, the Z field is used for a jump address. When SF = 17 and F = 0 or 3, the Z field is used for an IOB address. When F = 2, the Z field is loaded into the Shift Control register. These are the only specific actions taken by the hardware that affect the usage of the Z field. The hardware does nothing to prevent the Z field from being used for several things at once. For example, it could be used for a long constant and a jump address at the same time, or it could be used as an IO address and a jump address at the same time. The assembler, however, will flag an error if the programmer tries to load two different values into the same microinstruction field.

Constants

Constants can be 8 or 16 bits wide. If B = 1, the B input to the ALU is a constant. If F = 0 or 3 and SF = 0, the Y and Z fields form a 16 bit constant. If SF <> 0 and F <> 0 or 3 then Y is an 8 bit constant.

OldCarry

OldCarry (in ALU functions 15 and 17) is the carry from the immediately preceeding microinstruction, it is used for multiple precision arithmetic.

Condition Codes

All ALU related condition codes test the result of the ALU operation from the previous micro cycle. Thus the normal sequence is to perform an ALU operation and test its result in the next microinstruction. For example, comparison of two registers A and B could be done this way:

```
A - B;
if Gtr Goto(Label);    ! Jumps if A > B
```

All ALU tests with the exception of C19 test the lower 16 bits of the ALU. These are intended for data comparisons. After a subtraction, these condition codes compare the two operands. After other operations, these condition codes compare the 16-bit ALU result against zero.

C19 is designed for unsigned address comparisons. Assuming that A and B are registers containing 20-bit addresses and T is a temporary register, the following code fragments show how to compare A and B.

```
A - 1;
if C19 Goto(Label);    ! Jumps if A = 0, doesn't jump if A <> 0

T := A;
T := T - B;
T - 1;
if C19 Goto(Label);    ! Jumps if A = B, doesn't jump if A <> B

A - B;
if C19 Goto(Label);    ! Jumps if A < B, doesn't jump if A >= B

B - A;
if C19 Goto(Label);    ! Jumps if A > B, doesn't jump if A <= B
```

Memory Control

The memory system cycles in 680 ns (exactly 4 microcycles). Microcycles are numbered starting at 0 (t0, t1, t2 and t3). Requests must be made in a particular cycle (which cycle depends on the type of request). If a memory request is made in the wrong cycle, the processor will be suspended until the correct cycle. In some contexts, however, a request made in an improper cycle will be ignored--these contexts are explained below. There are 8 types of memory references, coded into the SF field when F = 1.

SF	Type	Description
16	Fetch	Fetch 1 word from Memory.
17	Store	Store 1 word into Memory
12	Fetch4	Fetch 4 words (0 mod 4 address)
13	Store4	Store 4 words (0 mod 4 address)
10	Fetch4R	Fetch 4 words, transport in reverse order
11	Store4R	Store 4 words, transport in reverse order
14	Fetch2	Fetch 2 words (0 mod 2 address)
15	Store2	Store 2 words (0 mod 2 address)

In the following discussion of the memory controller, the terms "Fetch" and "Store" are used for the memory functions which fetch or store exactly one word. The generic terms "fetch type" and "store type" are used for any type of fetching or storing memory reference.

The address for all memory references comes from R. For all fetch type references, the address (and the request itself) are latched at t3 and data is available from MDI or MDX (A = 3 or 4) at t2. If MDI or MDX is used during a t0 or t1 following a fetch type memory reference, the processor is suspended until t2.

Any address may be used with a Fetch, and the memory word may be read during any cycle from t2 until the following t1.

The address for a Fetch2 must be even (double-word aligned). If it is odd, the low-order bit of the address is ignored. After a Fetch2, the first word must be read at t2, and the second word must be read at t3.

The address for a Fetch4 must be quad-word aligned. If it is not quad-word aligned, the two low-order bits are ignored. After the Fetch4, the first word must be read at t2, and the succeeding words must be read at t3, t0, and t1.

Any address may be used with a Store. The address and Store command are given in a t2 cycle and the data to be written is supplied on R in the following t3.

The address for a Store2 must be even (double-word aligned). If it is odd, the low-order bit of the address is ignored. The address and Store2 are given in a t3 cycle, and the data is supplied on R in the following t0 and t1.

The address for a Store4 must be quad-word aligned. If it is not quad-word aligned, the two low-order bits are ignored. The address and Store4 are given in a t3 cycle, and the data is supplied in the next four cycles (t0, t1, t2 and t3).

The Fetch4R and Store4R types are identical to the Fetch4 and Store4 references except that word 3 of the quad word is received or sent first and word 0 last. (This is generally only useful for RasterOp so that it can do left to right as well as right to left transfers.)

Here are examples of each type of reference and how they are coded:

```
Fetch:      MA := Addr, Fetch;      (t3)
            ...                     (t0)
            ...                     (t1)
            Data := MDI;            (t2)

Fetch2:     MA := Addr, Fetch2;     (t3)
            ...                     (t0)
            ...                     (t1)
            Data0 := MDI;           (t2)
            Data1 := MDI;           (t3)

Fetch4:     MA := Addr, Fetch4;     (t3)
            ...                     (t0)
            ...                     (t1)
            Data0 := MDI;           (t2)
            Data1 := MDI;           (t3)
            Data2 := MDI;           (t0)
            Data3 := MDI;           (t1)

Fetch4R:    MA := Addr, Fetch4R;    (t3)
            ...                     (t0)
            ...                     (t1)
            Data3 := MDI;           (t2)
            Data2 := MDI;           (t3)
            Data1 := MDI;           (t0)
            Data0 := MDI;           (t1)

Store:      MA := Addr, Store;      (t2)
            MDO := Data;            (t3)

Store2:     MA := Addr, Store2;     (t3)
            MDO := Data0;           (t0)
            MDO := Data1;           (t1)

Store4:     MA := Addr, Store4;     (t3)
            MDO := Data0;           (t0)
            MDO := Data1;           (t1)
            MDO := Data2;           (t2)
            MDO := Data3;           (t3)
```

```
Store4R:    MA := Addr, Store4R;    (t3)
            MDO := Data3;           (t0)
            MDO := Data2;           (t1)
            MDO := Data1;           (t2)
            MDO := Data0;           (t3)
```

The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if both the processor and the IO system make memory requests, the IO is served first while the processor is delayed. The Hold bit, if set, locks out IO requests while it is set. To be effective, Hold must be asserted in a t2. This is necessary only when doing overlapped memory references. See the high performance rules below.

Combinations of memory references are tricky. There are a few rules which, if followed, will never let you go wrong, but they may preclude some clever twist of microcoding to save some cycles. The simple rules are:

1) Never start a memory reference after a fetch type reference until you have taken all the data.

2) Never start a memory reference during the four instructions which follow a store type request.

The full rules are complicated, but to achieve high-performance you need to consider them. The following rules define the way the memory controller treats memory requests.

1) After a Fetch or Fetch2 (in t3), any memory reference in t0 or t1 is ignored. A Store specified in the t3 will start immediately, but all others will abort until the correct time.

2) Fetch4 and Fetch4R follow the rules for Fetch and Fetch2 with the exception that a Store4 (in the same direction--forward or reverse) can be specified in t1, but this is only used for RasterOp.

3) After a Store (in t2), any memory reference in t3 or t0 is ignored. References started in t1 are aborted until the correct cycle.

4) After a Store2, Store4 or Store4R (in t3), any memory reference in t0 through t3 is ignored. Memory references started in t0 are aborted until the correct cycle.

5) To be effective, Hold must be asserted in a t2. You must be careful about aborts caused by using MDI in the wrong cycle--you may be aborted past the t2, causing the Hold to be ignored. You may not specify Hold too often--you must allow an IO reference at least once in every 3 memory cycles.

6) After a Fetch, MDI is valid from t2 through the following t1 (four full cycles). For Fetch2, Fetch4, and Fetch4R, each MDI is valid for a single microcycle.

Following these rules, we can construct many interesting overlapped memory requests. Note that in the following examples, Hold is always asserted in a t2. A Fetch ... Store sequence is an exception--you need not use Hold, but it doesn't hurt performance, so we assert it for consistency.

Indirect fetches:

```
MA := Addr, Fetch;                (t3)
instruction or Nop;               (t0)  must be explicit
instruction or Nop;               (t1)  must be explicit
MA := MDI, Fetch<n>, Hold;        (t2, t3)  any type of fetch
   ...
Data := MDI;                      (t2)
   ...
```

Hold is asserted in t2 so that IO requests do not pre-empt the processor. The instruction "MA := MDI, Fetch<n>, Hold;" first tries to execute in t2, but is aborted until t3 because it contains a fetch. The MDI is still valid because MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;                (t3)
instruction or Nop;               (t0)  must be explicit
instruction or Nop;               (t1)  must be explicit
instruction, Hold;                (t2)
MA := MDI, Fetch<n>;              (t3)  any type of fetch
   ...
Data := MDI;                      (t2)
   ...
```

Again, Hold is asserted in t2. Note that this differs from the previous example in that the Hold and Fetch<n> are not done in the same instruction. These two examples show that for indirect fetches, the two fetches may be separated by two or three other instructions.

Indirect stores:

```
MA := Addr, Fetch;                (t3)
instruction or Nop;               (t0)  must be explicit
instruction or Nop;               (t1)  must be explicit
MA := MDI, Store, Hold;           (t2)
MDO := Data;                      (t3)
```

In this case, the MDI, the Store, and the Hold all execute in t2.

```
MA := Addr, Fetch2;                (t3)
instruction or Nop;                (t0)  must be explicit
instruction or Nop;                (t1)  must be explicit
MA := MDI, Store, Hold; (t2)
MDO := MDI;                        (t3)
```

In this case, the first fetched word is used as an address, and the second is used as data to be stored.

```
MA := Addr, Fetch;                 (t3)
instruction or Nop;                (t0)  must be explicit
instruction or Nop;                (t1)  must be explicit
MA := MDI, Store<n>, Hold;         (t2, t3)  any except Store
MDO := Data;                       (t0)
   ...
```

Hold is asserted in t2 so that IO requests do not pre-empt the processor. The instruction "MA := MDI, Store<n>, Hold;" first tries to execute in t2, but is aborted until t3 because it contains a store. The MDI is still valid because MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;                 (t3)
instruction or Nop;                (t0)  must be explicit
instruction or Nop;                (t1)  must be explicit
instruction, Hold;                 (t2)
MA := MDI, Store<n>;               (t3)  any type except Store
MDO := Data;                       (t0)
   ...
```

Again, Hold is asserted in t2. Note that this differs from the previous example in that the Hold and Store<n> are not done in the same instruction. These two examples show that for indirect stores, the Fetch and the Store<n> may be separated by two or three other instructions.

Copy operations:

```
MA := Addr1, Fetch;                (t3)
instruction or Nop;                (t0)  must be explicit
instruction or Nop;                (t1)  must be explicit
MA := Addr2, Store, Hold;          (t2)
MDO := MDI;                        (t3)
```

A word is copied from one memory location to another. Unfortunately, two or four word copies are not possible because the times when data must be read and written are different for the fetches and stores.

Opcodes and operands

The OP file contains a 4-word sequence of instruction bytes. A quad-word address is contained in a XY register (UPC), and the 8 bytes pointed to by UPC are loaded into the OP file. The lower 3 bits of the byte address (byte within the quad word) are kept in BPC, a hardware register. BPC addresses the OP file to choose a byte. BPC is actually a 4-bit counter. It is incremented after each a byte is taken out of the OP file by NextInst (JMP=6, H=0) or NextOp (A=1). The 4th bit of BPC (BPC[3]), which is the "overflow" of the counter, is testable via a jump condition and indicates that all bytes in OP have been used.

The NextOp function (A=1) gets the next byte out of the instruction byte stream for use as an operand. The assembler automatically adds an "If BPC[3] GoTo(Refill)" jump clause. If BPC overflows, then control will go to Refill which increments UPC by 4, set BPC to 0, and starts a Fetch4 to the OP file. The special function LoadOp must be executed in the t1 after the Fetch4 to cause the Op file to be loaded with the data coming on MDI. Refill must then jump back to the instruction which needed the byte so that instruction may be re-executed. The instruction which executes NextOp must be capable of being executed twice (once when BPC overflowed and again when it is re-executed after Refill). This precludes instructions such as "R := NextOp + R".

In order for Refill to get back to the instruction which needs to be re-executed, the address of the failed NextOp is saved in a hardware register (Victim) if NextOp is executed when BCP[3] is set. The last instruction in Refill is coded with ReviveVictim (JMP=2, H=1), which sends control back to the "failed" NextOp.

BPC can be set without re-loading the OP file, and so the current quad word can be re-read without fetching it from memory a second time.

The NextInst jump enables a byte of the OP file (which is inverted for NextInst) into the Addr input of the micro-sequencer. The inverted byte is shifted left by 2 bits and OR-ed with ZAddr, sending control to address ZAddr + (OP' * 4). If BPC[3] is true, OP is forced to 377, sending control to location ZAddr, which is another version of Refill. This version of Refill also does the Fetch4 to the OP file, zeroes BPC, increments UPC by 4, and does the LoadOp, but then repeats the NextInst instead of returning via ReviveVictim.

To speed up the execution of Refill, the LoadOp Special Function loads all 4 words via hardware. The LoadOp should be given in the t1 following the Fetch4. The instruction which follows the LoadOp can go back to the NextInst/NextOp since the first byte is quaranteed to be in. The three remaining words arrive and are placed in OP by hardware without further microcode assistance. If BPC is set to a non-zero value (to start reading in the middle of the quad word), the Refill code must wait until the correct byte is in the OP file.

Shift Control

The PERQ shifter can rotate a 16 bit item 0 to 15 places and apply a mask to the shifter outputs. To use the shift hardware, the Z field of the instruction can be coded with the type of shift to be done with the F field set to F = 2. Coding of the shift control uses two 4 bit nibbles (shift control is inverted in the Z field):

Shift Control	Shift
0-17,0	1 bit field starting at bit 0-15
0-16,1	2 bit field starting at bit 0-14
0-15,2	3 bit field starting at bit 0-13
...	
0-2,15	14 bit field starting at bit 0-2
0-1,16	15 bit field starting at bit 0-1
0-17,17	Left shift 0-15
10-17,16	Rotate Right 8-15
10-17,15	Rotate Right 0-7
0-17,17-0	RightShift 0-15

The item to be shifted is placed on R, and the shifted and masked result can be read via SHIFT (A = 0) on the next instruction. The shift control logic keeps the last value loaded so that the shifter can shift a succession of words without respecifying the shift control function. The shift outputs always have the shifted value of what was last on R.

ShiftOnR

The ShiftOnR special function allows a shift function to be a variable. The shift control is obtained from the R bus and thus can be a data item. The usage sequence is: 1) Put the shift control (univerted)
on R and execute ShiftOnR, 2) Put the item to be shifted on R, and 3) Read the shifted result on SHIFT.

Expression Stack

The expression stack is used to evaluate expressions. Items are pushed on the stack by placing them on R and using the Push special function: "TOS := Data, Push". Items can be popped off the stack with the Pop special function. The top of the stack can be written without pushing or popping with the "TOS := Data" special function. The value on the top of the stack can be read at any time from TOS (A = 7). The stack is 16 levels deep. The stack can be reset (no items on the stack) by the StackReset special function. Stack empty can be read as a bit in USTATE.

Input/Output Bus

IOB is the input/output bus for PERQ. The IOB is a 16-bit bi-directional data bus plus a 7-bit address bus. The addresses are supplied on the Z Field. The eighth bit of the Z field indicates the direction of transfer (1=write, 0=read). To read an IO register, set SF = 1 and F = 0 or 3. The IO register is latched in the processor such that a succeeding microinstruction can read it from IOD (A = 2). IO registers can be written by putting a data value on R, putting the appropriate address in Z, and coding the IOB special function (SF = 1, F = 0 or 3).

Jumps

A jump needing an address normally gets it from the Z field. Since Z is only 8 bits wide and the control store is 4K, another 4 bits of address are needed. Short jumps branch to a location on the same 256-word page as the current microinstruction (CIA). To go to an arbitrary location, the F field can specify long jump (F = 3) which uses the SF field for the upper 4 bits of address.

The address for jumps might not come from the Z (and SF). Other sources for jump addresses are: 1) The S register (which is internal to the micro-sequencer), 2) A five level call stack (also internal to the micro-sequencer) which is pushed for a Call and popped for a Return, 3) The current instruction address plus 1, and 4) The Victim register.

There are three jumps which are multi-way branches. The three are: 1) NextInst, which is a 256-way branch based on a byte from the OP file; 2) Dispatch, which is a 16 way (or fewer) branch on the lower 4 bits of the SHIFT outputs; and 3) Vector, which branches to 1 of 8 micro-interrupt service routines. For all of these branches, the Z field of the micro-instruction supplies the other bits of the address. For NextInst, the resulting address is:

```
        0 0 0 0 0 0 0 0
Z Z P P P P P P P P Z Z
7 6 7 6 5 4 3 2 1 0 1 0
```

which results is a 256 way branch with a spacing of 4 instructions. For Dispatch, the address is:

```
Z Z Z Z Z Z S S S S Z Z
7 6 5 4 3 2 3 2 1 0 1 0
```

which also results in a spacing of 4 instructions. The Vector jump uses the outputs of the micro-interrupt priority encoder (V), which determines the highest priority micro-interrupt condition. The address is:

```
Z Z Z Z Z Z - V V V Z Z
7 6 5 4 3 2 0 2 1 0 1 0
```

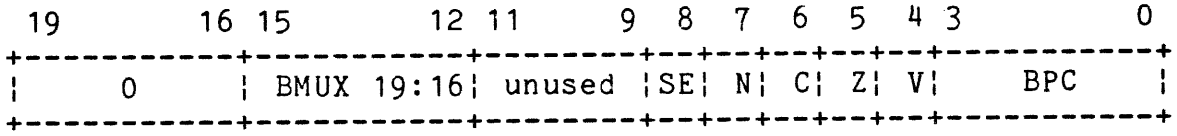
which also has a spacing of four instructions.

Interrupts

The hardware implements a microlevel interrupt which is used to allow the microprocessor to serve IO devices. There are (a maximum of) 8 interrupt requests which are assigned priorities by the hardware. When any of the interrupt requests is asserted, the branch condition IntrPend will succeed. The intended usage of this feature is that at convenient places in the microcode an instruction which has "If IntrPend Call(VecSrv)" is used. If any interrupts are pending, control will pass to VecSrv which should contain a Vector jump to send control to ZAddr + Vector*4 in the control store. The address of the interrupted instruction is on the call stack, and the interrupt microcode can serve the device and return like a subroutine would.

USTATE

The USTATE register contains various interesting items packed in a single word. The USTATE register (A=5) looks like:




```
BPC       Byte Program Counter
N         Negative (ALU result < 0)
Z         Zero (ALU result = 0)
C         Carry (ALU carry out of bit 15)
V         Overflow (ALU overflow occured)
SE        ESTK Empty (inverted data -- 0 = empty)
BMUX 19:16   Upper 4 Bits of BMUX, used to read bits 19:16
             a register (inverted data).  Do it this way:
                not UState(Register), Field(14,4);
                Result := Shift;
                ! Result[3:0] := Register[19:16]
```

Syntax of Micro-Programs

This describes the syntax of Perq micro-programs recognized by the PrqMic assembler. The syntax is described with a meta-language called EBNF (extended BNF). The following meta-symbols are used.

```
'     '     - surround literal text.
  ¦         - separate alternatives.
[     ]     - surround optional parts.
{     }     - surround parts which may be repeated
              zero or more times.
(     )     - are used for grouping.
  .         - ends a description.
```

```
name    = letter {letter ¦ digit} .

number  = ['2#' ¦ '8#' ¦ '10#' ¦ '#' ] digit {digit} .

constant        = name ¦ number .

register        = name .

label           = name .

empty   = .


MicroProgram    = {Instruction ';'} 'end' ';' .

Instruction     = {label ':'} Field {',' Field} .

    Field   = empty
            ¦ Pseudo
            ¦ PascalStyleConstant
            ¦ {Result ':='} ALU
            ¦ Jump
            ¦ Special .

        Pseudo  = 'Define' '(' name ',' constant ')'
                ¦ 'Constant' '(' name ',' constant ')'
                ¦ 'Opcode' '(' constant ')'
                ¦ 'Loc' '(' constant ')'
                ¦ 'Binary'
                ¦ 'Octal'
                ¦ 'Decimal'
                ¦ 'Nop'
                ¦ 'Case' '(' constant ',' constant ')'
                ¦ 'Place' '(' constant ',' constant ')' .

        PascalStyleConstant = name '=' constant .
```

```
Result  = register | SpecialResult .

    SpecialResult   = 'TOS' | 'MA' | 'MDO' | 'BPC'
                    | 'SrcRasterOp' | 'DstRasterOp'
                    | 'WidRasterOp' .

ALU     = ['not'] (Amux | Bmux)
        | Amux Op ['not'] Bmux
        | Amux '+' Bmux ['+' 'OldCarry']
        | Amux '-' Bmux ['-' 'OldCarry']
        | Amux 'amux' Bmux
        | Amux 'bmux` Bmux .

    Amux    = register | 'Shift' | 'NextOp' | 'IOD'
            | 'MDI' | 'MDX' | 'TOS'
            | 'UState' ['(' register ')'] .

    Bmux    = register | constant .

    Op      = 'and' | 'or' | 'xor' | 'nand' | 'nor'
            | 'xnor' .

Jump    = [Test] Goto ['(' Addr ')'] .

    Test    = 'If' Condition .

        Condition   = 'True' | 'False' | 'BPC[3]'
                    | 'C19' | 'IntrPend' | 'Odd'
                    | 'ByteSign' | 'Eql' | 'Neq'
                    | 'Gtr' | 'Geq' | 'Lss' | 'Leq'
                    | 'Carry' | 'OverFlow' .

    Goto    =  'Goto' | 'Call' | 'Return' | 'Next'
            | 'JumpZero' | 'LoadS' | 'GotoS' | 'CallS'
            | 'NextInst' | 'ReviveVictim' | 'PushLoad'
            | 'Vector' | 'Dispatch' | 'RepeatLoop'
            | 'Repeat' | 'JumpPop' | 'Loop'
            | 'ThreeWayBranch' .

    Addr    = label | constant .

Special = Nonary | Unary | Binary .

    Nonary  = 'WCSlow' | 'WCSmid' | 'WCShi' | 'LoadOp'
            | 'Hold' | 'StackReset' | 'Push' | 'Pop'
            | 'Fetch' | 'Fetch2' | 'Fetch4' | 'Fetch4R'
            | 'Store' | 'Store2' | 'Store4' | 'Store4R'
            | 'LatchMA' | 'ShiftOnR' .

    Unary   = UnaryName '(' constant ')' .

        UnaryName   = 'LeftShift' | 'RightShift'
                    | 'Rotate' | 'IOB'
                    | 'CntlRasterOp' .
```

```
Binary  = BinaryName '(' constant ',' constant ')' .

BinaryName  = 'Field' .
```

Notes on the Syntax

1) Programs are typed in free format. That is, a single micro-instruction may extend to as many lines as desired. Blank lines and lines consisting only of comments may be inserted anywhere. The exception to this rule is that a new micro-instruction begins with a new line, i.e. you may not place more than one instruction per line.

2) Names may be any length, but only 10 characters are significant when two names are compared.

3) Comments may be indicated by an exclamation mark: '!'. The remainder of the line following the exclamation mark is ignored. Comments may also be enclosed in braces Pascal style: '{' and '}' or '(*' and '*)'.

4) Numeric constants preceded by a '#' are octal constants.

5) Constants may be defined Pascal style:

```
name = value;
```

This allows including a file which contains constant definitions into both a Pascal program and a micro-program.

6) The following ALU functions are allowed by the syntax description, but do not exist. The assembler disallows them.

```
Amux 'nand' 'not' Bmux
Amux 'nor' 'not' Bmux
Amux 'xor' 'not' Bmux     (equivalent to Amux 'xnor' Bmux)
Amux 'xnor' 'not' Bmux    (equivalent to Amuv 'xor' Bmux)
```

7) The syntax allows constructions which are semantically incorrect. In other words, there are many combinations of actions which cannot be represented in a single instruction. For example,

```
TOS := MA := 10;          is valid, but

TOS := BPC := 10;         is invalid.
```

The 'Micro Instruction Format' section shows which fields of a micro-instruction are used by a particular action. The rule is that a certain field may be used only once. Thus since 'TOS :=' and 'BPC :=' both use the SF (special function) field, they both may not be used in a single micro-instruction.

8) Some goto types do not allow tests (are unconditional), and for some the test is optional. Similarly, some do not allow addresses, some require them, and for some the address field is optional. This table gives the rules.

req - required, opt - optional, <blank> - not allowed

Goto type	test	address
Goto	opt	req
Call	opt	req
Return	opt	
Next		
JumpZero		
LoadS		req
GotoS	opt	opt
CallS	opt	opt
NextInst		req
ReviveVictim		
PushLoad	opt	req
Vector	opt	req
Dispatch	opt	req
RepeatLoop		
Repeat		req
JumpPop	opt	req
Loop	opt	
ThreeWayBranch	opt	req

Assembler Commands

There are several commands which are directives to the assembler and placer to perform special actions. Assembler commands are indicated by a '$' in column 1. The entire line is considered to be an assembler command. You may not type other micro-instructions on the same line.

'$Include' FileName

The text in the named file is inserted into the microprogram as though it were present in the original source file. Included files may not be nested. Only the orginal source file may contain an Include command.

'$Title' TitleString

The TitleString is printed on the assembly listing on the first line of every page. The first Title command sets the main title which is printed at the left of each page. Each subsequent Title command sets the subtitle which is printed at the right of each page.

'$NoList'

The listing is turned off until a List command is encountered. This command has an effect only if a listing is requested when the placer is executed.

'$List' The listing is resumed if it was turned off by a NoList command. This command has an effect only if a listing is requested when the placer is executed.

Assembly Instructions

Before a micro-program can be run it must be assembled with PrqMic and then placed with PrqPlace. PrqMic translates the program into binary machine language, and PrqPlace assigns physical micro-store locations to those instructions which are not assigned by the micro-programmer.

The following shows how to assemble and place a micro-program. A micro-program source file name has the form <src>.MICRO. <src> is called the root name.

assemble:

```
PrqMic
Root file name? <src>

        -or-

PrqMic <src>
```

place with listing:

```
PrqPlace
Root file name? <src>
List file name? <lst>

        -or-

PrqPlace <src> <lst>
```

place without listing:

```
PrqPlace
Root file name? <src>
List file name?

        -or-

PrqPlace <src>
```

Once a micro-program has been assembled and placed, you can use it in one of these ways:

a. Load it into another Perq with OdtPrq.

b. Load it into the same Perq with the ControlStore module.

c. Write it into a boot file with MakeBoot.

Writable Control Store (WCS) Map

The following provides a map of the Writable Control Store (WCS) and describes the micro code register usage.

When you boot the system, VFY.MICRO runs memory diagnostics. SYSB.MICRO then loads microcode from the MBoot file. At boot time, microcode can only be loaded into WCS addresses 0 through 7377. At the end of the boot, only PERQ.MICRO and IO.MICRO are in the WCS. This leaves 7000 through 7777 for user defined microcode. (You can use 7000 through 7377 for bootable special purpose microcode.)

PERQ.MICRO -PERQ Q-Code interpreter microcode. Temporary registers store state information during Q-Code interpretation. Registers: 3-21, 51-57, 64-67, 370. Temporary registers: 30-50, 61-63, 70-77. Placement: 0-4377.

IO.MICRO - Input/Output microcode. Registers: 200-207, 211-217, 221-233, 235-252, 255-260, 262-266, 276, 277, 327, 373, 374. Temporary registers: 220, 261. Placement: 4400-5777.

LINK.MICRO - 16 bit parallel interface microcode. (Not normally booted into WCS). Registers: 350, 351. Placement: 6744-7400.

SYSB.MICRO - system boot microcode. Registers: 0-64. Placement: 7000-7777.

IOE3.MICRO - Microcode for 3MBaud ETHERNET. This area is available if the system will not use ETHERNET. Registers: 270-274. Placement: part of IO.MICRO.

BOOT.MICRO - Boot microcode. Registers: 0-2, 4, 10, 20, 40, 100, 200, 252, 277, 307, 337, 350, 357, 367, 373, 375, 376, 377. Placement: 0-777.

KRNL.MICRO - PERQ microcode kernel. Registers: 0, 357-377. Placement: 7400-7777.

GOODBY.MICRO - Power down microcode. Placement: 5000-5377.

ETHER10.MICRO - 10MBaud ETHERNET microcode. Registers: 300-321. Placement: 7000-7300

RO.MICRO - Raster-op microcode. Registers: 100-124, 130-142, 370. Placement: part of PERQ.MICRO.

LINE.MICRO - Line drawing microcode. Registers: 100, 101, 103-116. Placement: part of PERQ.MICRO.

VFY.MICRO - Verifies that the hardware seems to work. Registers: 0-16, 300, 301, 370. Placement: 4000-6377.

Quirks

As of 12/1/80, the following quirks are known:

- The Z field is inverted for Shift functions (assembler fixes this).
- The Op file is inverted on NextInst (assembler Opcode does it).
- The Z field is inverted for all addresses (assembler fixes it).
- IOB functions are executed twice if an abort occurs.
- C19 will not be valid if an abort occurs on the test.
- C19 test is inverted sense (i.e. jump if no carry out of bit 19).
- UState[15:12] (the upper BMux bits) are inverted.
- Condition codes are not quite right after double precision adds and subtracts. See the `Condition Codes` section.
- Condition codes are invalid after ReadProduct, ReadQuotient, and ReadVictim.

PERQ QCode Reference Manual

Miles A. Barel
John P. Strait

June 25, 1981

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Table of Contents

Table of Contents

1. Q-Machine Architecture

1.A Definitions

Segment - A segment is the underlying structure of PERQ's virtual memory system. It is the largest area of contiguous memory, and also the unit of swappability. Segments come in two types: code segments, which are byte-addressed, read-only, and fixed in size with a maximum size of 64K bytes (32K words); and data segments, which are word-addressed, read-write, and variable in size with a maximum size of 64K words.

MSTACK - Memory Stack. A data segment which contains the user run-time stack.

ESTACK - Expression Stack. A 16 level expression evaluation stack (internal to the PERQ processor).

MTOS - Top of MSTACK. MTOS refers to the virtual address of the top of the memory stack. (MTOS) denotes the item on the top of the MSTACK.

ETOS - Top of ESTACK. (ETOS) denotes the item on the top of the ESTACK.

Activation Record - Stack segment fragment for a single routine containing local variables, parameters, function result, temporaries (anonymous variables), other housekeeping values (Activation Control Block - defined below), and a copy of the EStack at the time the activation record is created.

CB - Code Base (register). Physical adddress of the base of the current code segment.

SB - Stack Base (register). Physical address of the base of the current stack segment.

PC - Program Counter (register). Physical address of the current instruction.

GDB - Global Data Block. A GDB contains the global variables for a particular module. GDBs always begin on a double-word boundary.

ISN - Internal Segment Number (compiler-generated).

SSN - System Segment Number (system-generated). Note, System Segment 0 is reserved and may never be used.

LL - Lexical Level. Note: the Lexical Level of the main body of a process is always 0.

RN - Routine Number (register). RN contains the ordinal number of the current routine. Note: RN must lie in the range 0 to 255.

CS - Code Segment (register). CS contains the system segment number (SSN) for the current code segment. This segment must be resident in physical memory for a process to be runnable.

SS - Stack Segment (register). SS contains the system segment number (SSN) for the current stack segment. This segment must be resident in physical memory for a process to be runnable.

PS - Parameter Size. PS is the number of words in an activation record which are used for parameters.

RPS - Result + Parameter Size. This is the number of words in an activation record which are used for function result and parameters.

LTS - Local + Temporary Size. LTS is the number of words in an activation record which are used for locals and temporaries (anonymous variables). (Note: the LTS of a main program body is always forced to 0.)

AP - Activation Pointer (register). AP contains the physical address of the current activation record.

DL - Dynamic Link. This is the AP of the caller, represented as an offset from SB.

SL - Static Link. This is the AP of the surrounding routine, represented as an offset from SB.

TP - Top Pointer (register). TP contains the physical address of the top of the run-time MStack.

TL - Top Link. TP of the caller, represented as an offset from SB.

GP - Global Pointer (register). Physical address of the GDB for the current code segment.

GL - Global Link. GP of the caller, represented as an offset from SB.

LP - Local Pointer (register). Physical address of the current activation record. When the LP is stored in an Activation Control Block (ACB), it is represented as an offset from SB. Unlike other values in the ACB, the LP value is the current value of the Local Pointer, not some previous value.

XGP - eXternal Global Pointer. Pointer to another code segment's GDB, represented as an offset from SB.

XST - eXternal Segment Table. For a given program module, the XST translates ISNs to SSNs and XGPs.

RS - Return Segment. RS is the CS of the caller.

RA - Return Address. PC of the caller, represented as an offset from CB.

RR - Return Routine. RN of the caller.

RD - Routine Dictionary. Each code segment contains a routine dictionary which is indexed by RN. For each routine, the routine dictionary gives the lexical level (LL), entry address, exit address, parameter size (PS), result + parameter size (RPS), and local + temporary size (LTS).

ACB - Activation Control Block. The ACB contains housekeeping values in the activation record. It contains the SL, LP, DL, GL, RS, RA, RR and EP. In the ACB, the DL, GL, RS, RA, and RR are the AP, GP, CS, PC, and RN of the caller, respectively. The SL is the AP of the routine that surrounds the current one. The LP in the ACB is the current local pointer.

EEB - Exception Enable Block - Each EEB enables a single exception by associating an exception with a handler. A (possibly empty) list of EEBs is associated with each activation record in the stack.

Enabling an Exception - Associating a certain exception handler with a certain exception.

EP - Exception Pointer. The address (as an offset from SB) of a list of nodes that describe which exceptions are enabled in a certain routine.

ER - Exception Routine Number. The routine number of an exception.

ES - Exception Segment Number. The segment number of an exception.

Exception - An error or unusual occurrence in the execution of a routine or program.

Exception Handler - A procedure to be executed when a certain exception is raised.

HR - Handler Routine Number. The routine number of an exception handler.

NE - Next Exception. The address (as an offset from SB) of the next in a list of nodes that describe which exceptions are enabled in a certain routine.

Raising an Exception - Asserting a certain exception.

1.B Memory Organization

The PERQ's virtual memory system features a segmented 32-bit virtual address space mapped into a 20-bit physical address space. The segment is the unit of swappability, and comes in two types:

1) Code segments which are byte-addressed, read-only, and fixed in size with a maximum size of 64K bytes (32K words).

2) Data segments which are word-addressed, read-write, and variable in size with a maximum size of 64K words.

A PERQ process is a collection of up to 64K code and data segments. One of the data segments is the stack segment. Every process must have a stack segment and at least one code segment.

All segments are allocated in 256 word chunks and when in physical memory are aligned on 256 word boundaries. Note: A single segment must exist in contiguous memory. It may not be fragmented.

1.B.1 Memory Organization at the Process Level

The memory organization is designed with the following attributes in mind: 1) to allow separately compiled code segments to be grouped into a single process, 2) to allow code segments to be shared among processes, 3) to allow each code segment to have its own global variables, and 4) to allow one code segment to reference routines and global variables in other code segments. To achieve this, the following high-level characteristics are implemented:

1) All code is re-entrant.

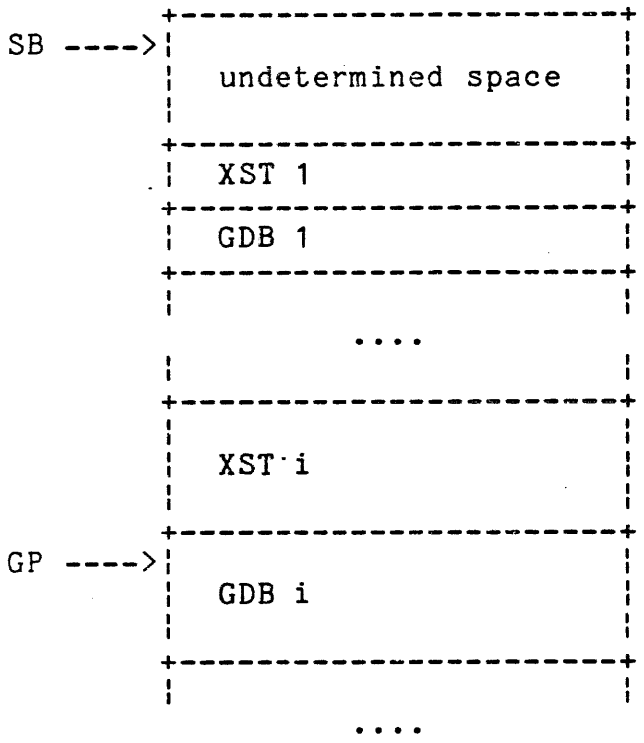
2) Each code segment only refers to other code segments by internal (compiler-generated) segment numbers, which are not necessarily the same as the system-assigned segment numbers.

3) Each code segment in a process has its own Global Data Block on the run-time stack.

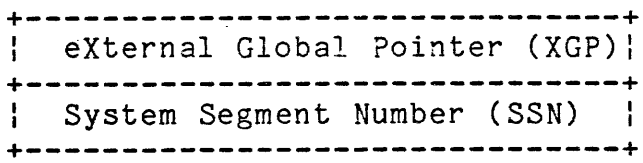
4) Each code segment has an external segment table to permit referencing global variables and routines from other code segments.

1.B.1.a Global Data

At the global level, there is a Global Data Block (GDB) and an eXternal Segment Table (XST) associated with each code segment in a process. For a particular program module, the GDB contains the global variables, and the XST translates internal (compiler-generated) segment numbers (ISNs) to actual system segment numbers (SSNs) and eXternal Global Pointers (XGPs). To simplify the system, we devote a single pointer to reference both the current GDB and XST. This Global Pointer (GP) points to the lowest address in the GDB and is ALWAYS aligned on a double-word boundary.

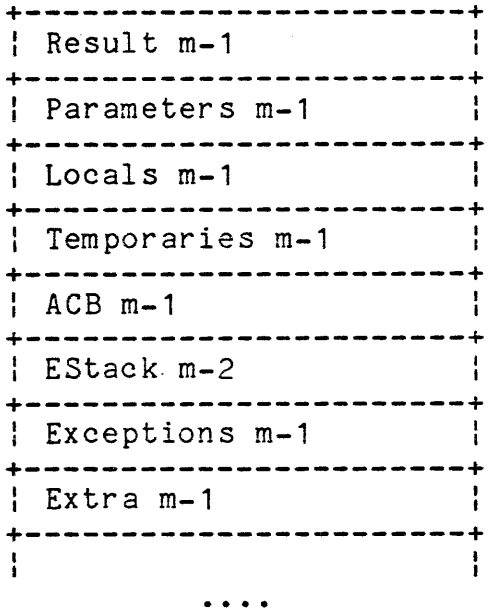


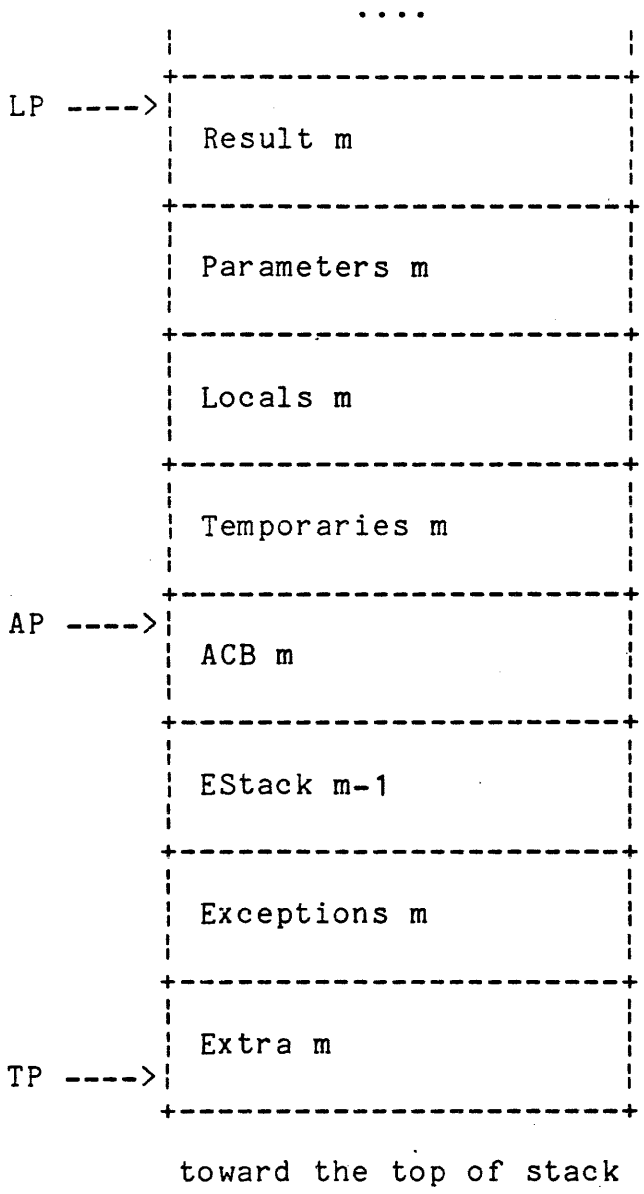
The XST for each segment is indexed by the internal segment numbers (ISNs). The entry is at GP - 2*ISN (Note: There is no entry for ISN 0; ISN 0 always refers to the current segment). Each entry contains the offset from stack base (SB) of an external data block (XGP) and the actual system segment number (SSN) of the external segment. The XGP values are set by the linker, and the SSN values are set by the loader.



1.B.1.b Local Data

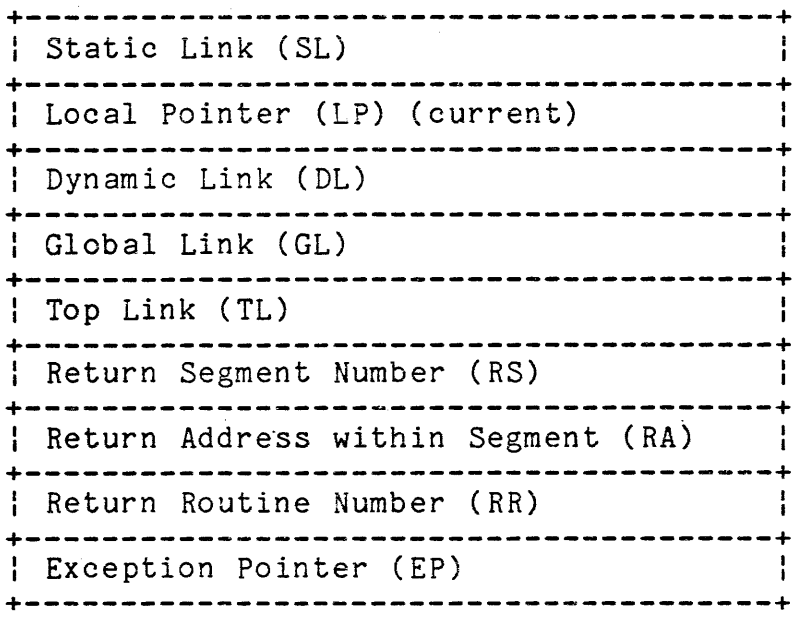
At the local level, there is an activation record, which consists of local variables, function result, parameters, temporaries (anonymous variables), the Activation Control Block (ACB), the previous EStack, exception enable blocks, and extra values that the routine may push and pop from the run-time stack. Three pointers are used to access and keep track of this information: the top-of-stack pointer (TP), the current-activation pointer (AP), and the local-variables pointer (LP).





The function result, parameters, locals and temporaries are located by an offset from LP.

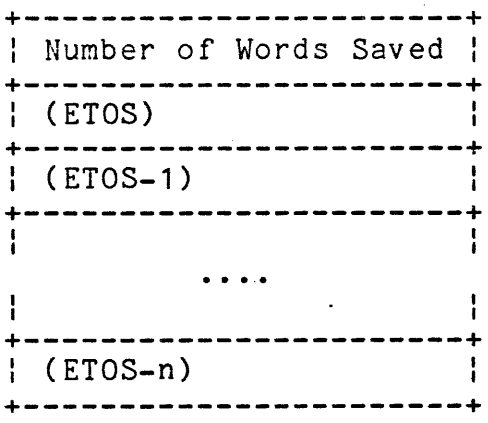
Each ACB has the following form:



toward the top of stack

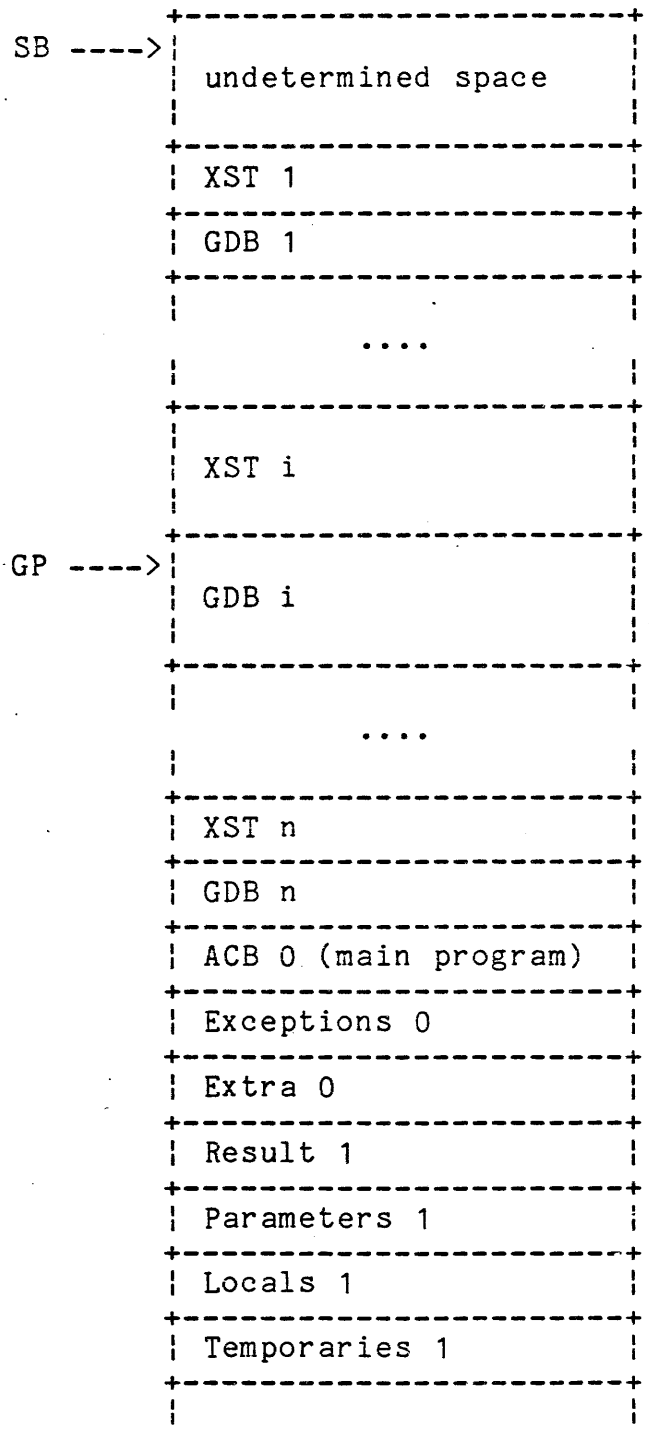
The values in the ACB are the AP of the surrounding routine (SL), the current (not previous) LP, the AP of the caller (DL), the GP of the caller (GL), the TP of the caller (TL), the SSN of the caller (RS), the program counter (PC) of the caller (RA), the RN of the caller (RR) and a pointer to the current exception enable records (EP). Note: When previous pointer values are saved in the ACB they are called links: SL, DL, GL, TL. Because the current (not previous) LP and EP are stored in the ACB, they are called pointers, not links. The static link is not used for main programs and top-level routines. It is zero for these routines and is therefore a means of detecting top-level routines. The dynamic link is zero for the first routine on the stack (the one at the base of the stack). This is used to detect the end of the stack during stack searches.

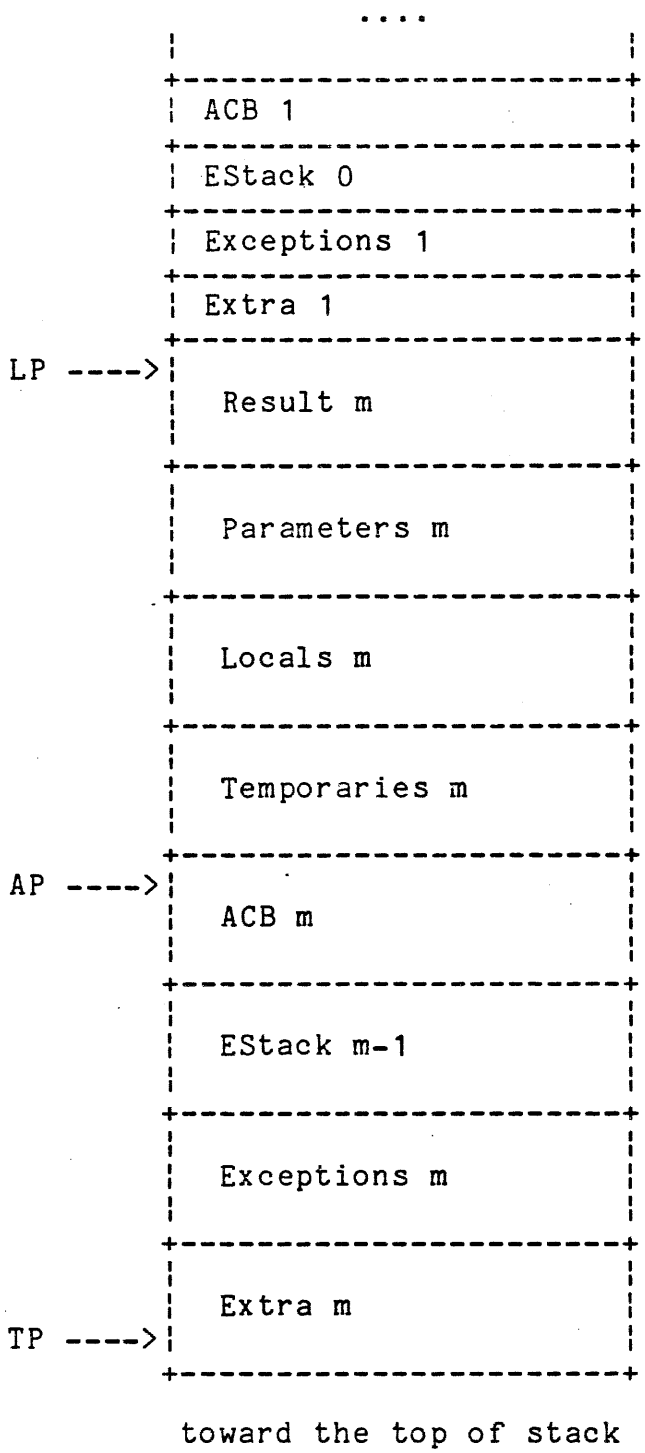
The EStack image immediately follows the ACB and looks like this:



1.B.1.c Run-Time Stack Organization

The following is an outline of the stack for a process of n segments, executing the mth routine call, which is in the ith segment:




....
ACB 1
EStack 0
Exceptions 1
Extra 1
LP ---->
Result m
Parameters m
Locals m
Temporaries m
AP ---->
ACB m
EStack m-1
Exceptions m
Extra m
TP ---->
toward the top of stack

1.B.2 Memory Organization at the System Level

The system makes use of two tables to control memory usage, the System Segment Address Table and the System Segment Information Table. The former contains all information which is needed by the Q-Code micro-code (location, size, residency, etc). The latter contains other information which is only referenced by the operating system (reference, I/O and lock counts; maximum size; etc).

1.B.2.a System Segment Address Table

The System Segment Address Table is a dynamic table, which is always resident in physical memory starting at physical address 0. This table contains two words per segment, and contains all information that the Q-Code micro-code needs to know about each segment. The information contained in this table is:

1) Segment Base Address (upper 12 bits)

2) Segment Size (number of 256 word blocks - 1)

3) Flags
   - Not Resident
   - Recently Used
   - Moving
   - Shareable
   - Segment Kind
   - Segment Full
   - Segment Table Entry In Use

The Segment Base Address is the upper 12 bits of the physical address of the base of the segment. If the segment is not resident in physical memory, this field is undefined. The lower 8 bits of the Segment Base Address are always guaranteed to be zero (since all segments are aligned on 256-word boundaries).

The Segment Size is one less than the size of the segment in 256-word blocks (i.e., Segment Size 0 = 256 words).

The Flags have the following meanings and uses:

Not Resident - When true, this flag indicates that the segment is either swapped out or that the segment table entry is not in use. When false, this flag indicates that the entry is in use and the segment it describes is resident in physical memory. (See the "Segment Table Entry In Use" flag.)

Recently Used - This flag is set when a segment is accessed. It is used by the swapper to determine which segments are likely candidates to be swapped out when space is needed.

Moving - This flag, when true, indicates that the segment is being moved from one location in physical memory to another. If moving is true, Resident will be false. Moving is used only by the swapper to determine how to handle segment faults. (Not used by the Q-Code micro-code).

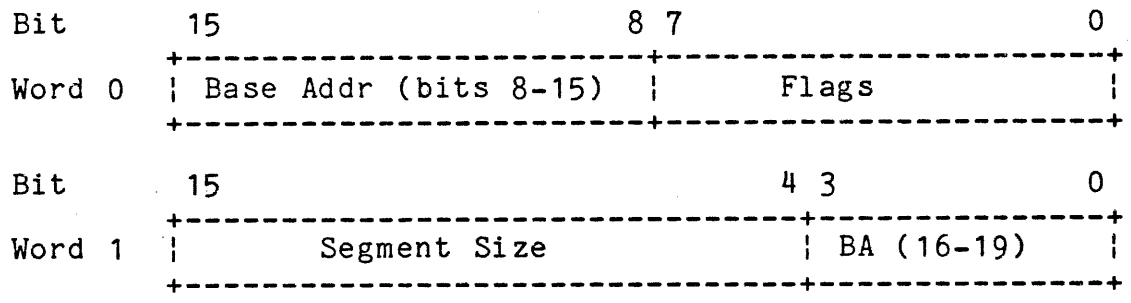
Shareable - When true, this flag indicates that a segment may be shared by several processes. (Not used by the Q-Code micro-code)

Segment Kind - This flag indicates whether the segment is a data or code segment. (Not used by the Q-Code micro-code)

Segment Full - This flag, when true, indicates that the entire data segment has been allocated (via the Pascal New procedure). This flag is needed to distinguish full and empty data segments (and has no relevant meaning for code segments). (Not used by the Q-Code micro-code)

Segment Table Entry In Use - This flag is set true when the segment table entry describes a valid segment.

The arrangement of these fields within the two words are shown below:



The positions of the flags within the low byte of Word 0 are:

Bit	Flag
0	Resident
1	Moving
2	Recently Used
3	Shareable
4	Segment Kind
5	Segment Full
6	Table Entry In Use
7	not used

1.B.2.b System Segment Information Table

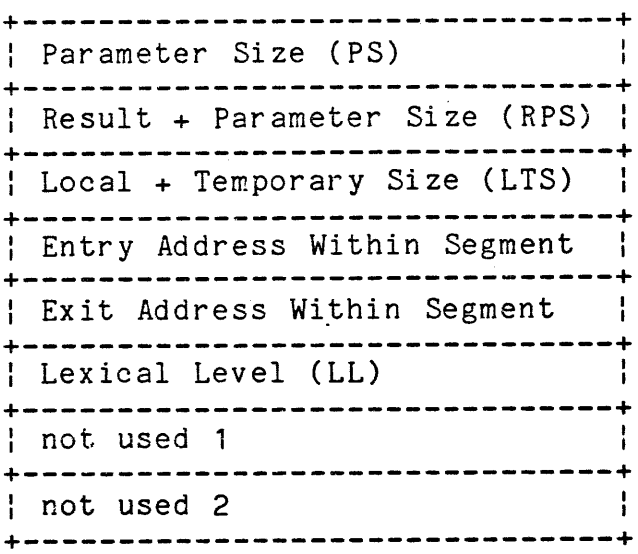
There is no information in the System Segment Information Table which is needed by the Q-Code micro-code; hence it is not described here. See "Module Memory" in the Operating System Manual.

1.B.2.c Code Segment Organization

A code segment contains the code for all routines in a segment and a routine dictionary which contains vital information about each of these routines.

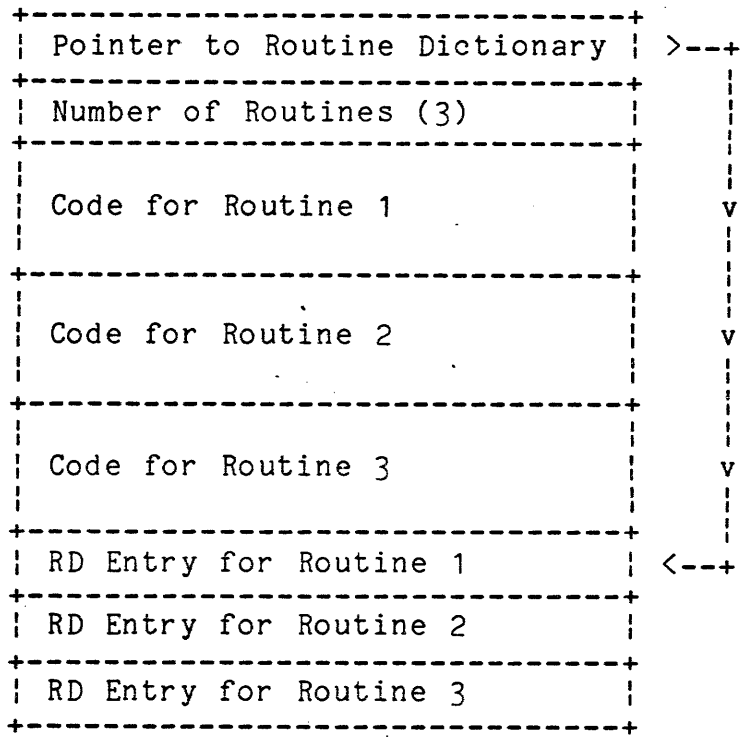
The first word of every code segment is the offset from the base of the segment to the first word of the routine dictionary. The second word contains the number of routines which are defined in the segment. These two words are followed by the actual code which comprise the routines. Finally, the code is followed by the routine dictionary. The code is padded with 0 to 3 words of zero (by the compiler) so that the routine dictionary is aligned on a quad-word boundary. This is possible since the compiler knows that the base of the segment is also aligned on a quad-word boundary. It should also be noted that each entry in the dictionary is exactly 2 quad-words long (8 words). The routine dictionary is indexed by (Base Address of Dictionary)+8*RN.

Each entry has the following form:



The Entry and Exit Addresses are the offsets from code base (CB) to the beginning of the routine and the beginning of the "terminate code" of the routine.

The following is a sample of a code segment containing 3 routines:



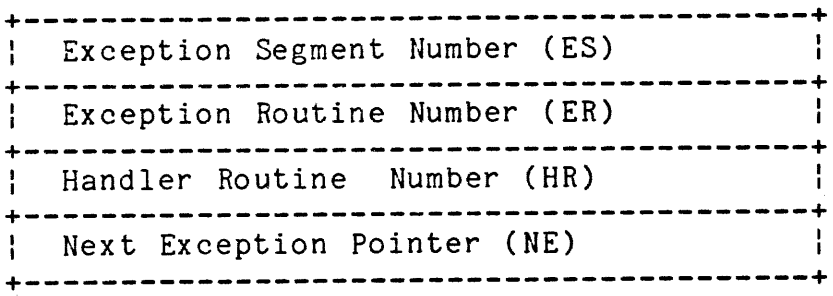
1.C Error Handling and Fault Conditions

Error processing is done through an exception handling mechanism. The syntax of exception and handler is described in the PERQ Pascal Extensions manual. The QCodes used to enable and raise exceptions are described in Section "Routine Calls and Returns."

An exception is declared in a way which is similar to a procedure declaration. Therefore it is convenient to assign a routine number at the point of definition of an exception. There is a corresponding entry in the routine dictionary to describe this exception. This entry describes a procedure with the correct number of parameters along with no locals or temporaries. The system segment number of the module, which contains the exception, and its routine number uniquely identify the exception.

An exception handler is simply a procedure--handlers may not return a value. An exception handler is enabled by declaring it inside some routine. This outer routine is called the enabler of the handler. The code segment numbers and global pointers of the enabler and handler are the same. The static link of the handler is the same as the activation pointer of the enabler. Thus an exception handler is uniquely identified by the ACB of the enabler and the routine number of the handler.

An exception enable record consists of the definition of the exception to be handled (ER and ES), the routine number of the handler (HR), and a link to the next exception enable record (NE). ER and ES are negative for a handler of all exceptions.



When a routine is called, the exception pointer (EP) within the new ACB is set to zero to indicate that there are no exception handlers. If exception handlers are declared within the routine, the compiler generates appropriate QCodes to add those handlers to the routine's exception list. When a routine is exited, the exception records are popped from the run-time stack along with the rest of the activation record.

When an exception is raised, the QCode interpreter microcode calls the procedure RaiseP in the module Except. This routine searches back through the run-time stack to find the most recent routine which contains a handler for that exception. Once such a candidate is found, the stack is searched again to determine if that handler is already active. If it is, the search for a candidate continues. This implementation ensures that while a particular instance of a handler is active, it will not be activated again. A recursive routine may contain a handler, and there may be several instances of the same handler. In this case, each handler is activated separately.

Determining if a handler is active by searching the stack is not the best method. If we assume that the depths of handler activation records are related to the number of active handlers, then the total stack search time for raising an exception is related to the square of the number of active handlers of that exception. This should rarely be burdensome because it is not expected that there will be more than one or two active handlers for a given exception. Recursive routines that pass exceptions up the call stack are pathological cases.

When an unused enable for the exception is found, the associated handler is called. The handler has the option of exiting to the routine which enabled the exception (via a Goto) or of returning to the point where the exception was raised (by falling off the end of the procedure).

If no handler is found, the default handler is called. The search order for a certain routine's exception list is the reverse of the order in which they were enabled. If a handler of all exceptions is declared within a certain routine, the compiler enables it first.

A handler of all exceptions is called in a special way. When it is called, the parameters that were passed when the exception was raised have already been pushed onto the stack. A new activation record for the handler is built above those parameters. Such a handler for all exceptions must have four words of parameters: the segment and routine numbers of the exception that was raised, a pointer to the first word of the original parameters, and a pointer to the first word after the original parameters. The two pointers are represented as integer offsets from the base of the stack.

2. Instruction Format

Instructions on the Q-machine are one byte long followed by zero to four parameters. Parameters are either a signed byte (B : range -128 to 127), an unsigned byte (UB : range 0 to 255) or a word (W). Words need not be word aligned (unless specified). The low byte is first in the instruction byte stream.

Any exceptions to these formats are noted with the instructions where they occur.

3. Pointers

There are five different types of pointers, defined as follows: (Note: 20-bit offsets may only exist on the EStack).

Word Pointer: A 20-bit offset from StackBase (StackBase is the 20 bit physical address of the base of the stack).

Byte Pointer: A 20-bit offset from StackBase to the base of the byte array (TOS-1) and a byte offset into the array (TOS).

String Pointer: Same as a byte pointer.

Packed Field Pointer: A 20-bit offset from StackBase to the base of the word the field is in (TOS-1) and a one word field descriptor (TOS).

Field Descriptor:
Bits 0-3: The field width (in bits) minus 1

Bits 4-7: The rightmost bit of the field.

Pascal Pointer: Obtained by declaring a variable as a pointer to another data type (i.e., var I: ^Integer;). (TOS-1) is the system segment number that contains the datum. (TOS) is the offset from the segment base to the datum.

Implementation Note: Stacks grow from low addresses to high addresses (i.e., if the address of TOS is 10 then the address of TOS-1 is 9 -- not 11).

4. QCode Descriptions

This section provides a detailed description for each QCode. The QCode descriptions appear categorically. The following lists the QCodes and equates each with its respective QCode number. Thus, if you only know the QCode number, you can use the list to equate the number to the named QCode.

QCode OpCode Definitions:

```
LDC0      = 0; (Assignment of Byte/Word opcodes are important)
LDC1      = 1;
LDC2      = 2;
LDC3      = 3;
LDC4      = 4;
LDC5      = 5;
LDC6      = 6;
LDC7      = 7;
LDC8      = 8;
LDC9      = 9;
LDC10     = 10;
LDC11     = 11;
LDC12     = 12;
LDC13     = 13;
LDC14     = 14;
LDC15     = 15;
LDCMO     = 16;
LDCB      = 17;
LDCW      = 18;
LSA       = 19;
ROTSHI    = 20;
STIND     = 21;
LDCN      = 22;
LDB       = 23;
STB       = 24;
LDCH      = 25;
LDP       = 26;
STPF      = 27;
STCH      = 28;
EXGO      = 29;
QAND      = 30;
QOR       = 31;
QNOT      = 32;
EQUBool   = 33; (Opcode assignment of all EQU,NEQ,LEQ,LES)
NEQBool   = 34; (GEQ and GTR qcodes are important
LEQBool   = 35;
LESBool   = 36;
GEQBool   = 37;
GTRBool   = 38;
```

```
EQUI      = 39;
NEQI      = 40;
LEQI      = 41;
LESI      = 42;
GEQI      = 43;
GTRI      = 44;
EQUStr    = 51;
NEQStr    = 52;
LEQStr    = 53;
LESStr    = 54;
GEQStr    = 55;
GTRStr    = 56;
EQUByt    = 57;
NEQByt    = 58;
LEQByt    = 59;
LESByt    = 60;
GEQByt    = 61;
GTRByt    = 62;
EQUPowr   = 63;
NEQPowr   = 64;
LEQPowr   = 65;
SGS       = 66; (there is no LESPowr
GEQPowr   = 67;
SRS       = 68; (there is no GTRPowr
EQUWord   = 69; (Word is the last comparison and only EQU
NEQWord   = 70; and NEQ exist)
ABI       = 71;
ADI       = 72;
NGI       = 73;
SBI       = 74;
MPI       = 75;
DVI       = 76;
MODI      = 77;
CHK       = 78;
INN       = 88;
UNI       = 89;
QINT      = 90;
DIF       = 91;
EXITT     = 92;
NOP       = 93;
REPL      = 94;
REPL2     = 95;
MMS       = 96;
MES       = 97;
LVRD      = 98;
LSSN      = 99;
XJP       = 100;
RASTOP    = 102;
STRTIO    = 103;
INTOFF    = 105;
```

```
INTON     = 106;
LDLB      = 107;
LDLW      = 108;
LDL0      = 109;
LDL1      = 110;
LDL2      = 111;
LDL3      = 112;
LDL4      = 113;
LDL5      = 114;
LDL6      = 115;
LDL7      = 116;
LDL8      = 117;
LDL9      = 118;
LDL10     = 119;
LDL11     = 120;
LDL12     = 121;
LDL13     = 122;
LDL14     = 123;
LDL15     = 124;
LLAB      = 125;
LLAW      = 126;
STLB      = 127;
STLW      = 128;
STL0      = 129;
STL1      = 130;
STL2      = 131;
STL3      = 132;
STL4      = 133;
STL5      = 134;
STL6      = 135;
STL7      = 136;
LDOB      = 137;
LDOW      = 138;
LDO0      = 139;
LDO1      = 140;
LDO2      = 141;
LDO3      = 142;
LDO4      = 143;
LDO5      = 144;
LDO6      = 145;
LDO7      = 146;
LDO8      = 147;
LDO9      = 148;
LDO10     = 149;
LDO11     = 150;
LDO12     = 151;
LDO13     = 152;
LDO14     = 153;
LDO15     = 154;
LOAB      = 155;
```

```
LOAW      = 156;
STOB      = 157;
STOW      = 158;
STO0      = 159;
STO1      = 160;
STO2      = 161;
STO3      = 162;
STO4      = 163;
STO5      = 164;
STO6      = 165;
STO7      = 166;
MVBB      = 167;
MVBW      = 168;
MOVB      = 169;
MOVW      = 170;
INDB      = 171;
INDW      = 172;
IND0      = 173;
IND1      = 174;
IND2      = 175;
IND3      = 176;
IND4      = 177;
IND5      = 178;
IND6      = 179;
IND7      = 180;
LDIND     = 173; (Same as IND0))
LGAWW     = 181;
STMW      = 182;
STDW      = 183;
SAS       = 184;
ADJ       = 185;
CALLL     = 186;
CALLV     = 187;
ATPB      = 188;
ATPW      = 189;
WCS       = 190;
JCS       = 191;
LDGB      = 192;
LDGW      = 193;
LGAB      = 194;
LGAW      = 195;
STGB      = 196;
STGW      = 197;
RETURN    = 200;
MMS2      = 201;
MES2      = 202;
LDTP      = 203;
JMPB      = 204;
JMPW      = 205;
JFB       = 206;
```

```
JFW       = 207;
JTB       = 208;
JTW       = 209;
JEQB      = 210;
JEQW      = 211;
JNEB      = 212;
JNEW      = 213;
IXP       = 214;
LDIB      = 215;
LDIW      = 216;
LIAB      = 217;
LIAW      = 218;
STIB      = 219;
STIW      = 220;
IXAB      = 221;
IXAW      = 222;
IXA1      = 223;
IXA2      = 224;
IXA3      = 225;
IXA4      = 226;
TLATE0    = 227;
TLATE1    = 228;
TLATE2    = 229;
EXCH      = 230;
EXCH2     = 231;
INCB      = 232;
INCW      = 233;
CALLXB    = 234;
CALLXW    = 235;
LDMC      = 236;
LDDC      = 237;
LDMW      = 238;
LDDW      = 239;
STLATE    = 240;
QLINE     = 241;
ENABLE    = 242;
QRAISE    = 243;
LDAP      = 244;
ROPS      = 250; (See below for 2nd byte)
INCDDS    = 251;
LOPS      = 252; (See below for 2nd byte)
BREAK     = 254;
ReFillOp  = 255;
```

Real Operations - Second byte of ROPS opcode:

```
TNC       = 0;
FLT       = 1;
ADR       = 2;
NGR       = 3;
```

```
SBR       = 4;
MPR       = 5;
DVR       = 6;
RND       = 7;
ABR       = 8;
EQUReal   = 9;
NEQReal   = 10;
LEQReal   = 11;
LESReal   = 12;
GEQReal   = 13;
GTRReal   = 14;
RUNUSED   = 15;
```

Long Operations - Second byte of LOPS opcode:

```
CVTLI     =  0;
CVTIL     =  1;
ADL       =  2;
NGL       =  3;
SBL       =  4;
MPL       =  5;
DVL       =  6;
MODL      =  7;
ABL       =  8;
EQULong   =  9;
NEQLong   = 10;
LEQLong   = 11;
LESLong   = 12;
GEQLong   = 13;
GTRLong   = 14;
LUnused   = 15;
```

4.A Variable Fetching, Indexing, Storing and Transferring

4.A.1 Loads and Stores of One Word

4.A.1.a Constant One Word Loads

LDC0..15	0-15	Load Word Constant. Pushes the value (0..15), with high byte ero, onto the EStack.
LDCN	22	Load Constant Nil. Pushes the value of NIL onto the EStack.
LDCM0	16	Load Constant -1.

LDCB	B	17	Load Constant Byte. Pushes the next byte on the EStack, with sign extend.
LDCW	W	18	Load Constant Word. Pushes the next word on the EStack.

4.A.1.b Local One Word Loads and Stores

LDL0..15		109-124	Short Load Local Word. LDLx fetches the word with offset x in the current activation record and pushes it onto the EStack.
LDLB	UB	107	Load Local Word/Byte Offset. Fetches the word with offset UB in the current activation record and pushes it on the EStack.
LDLW	W	108	Load Local Word/Word Offset. Fetches the word with offset W in the current activation record and pushes it on the EStack.
LLAB	UB	125	Load Local Address/Byte Offset. Pushes a word pointer to the word with offset UB in the current activation record on EStack.
LLAW	W	126	Load Local Address/Word Offset. Pushes a word pointer to the word with offset W in the current activation record on EStack.
STL0..7		129-136	Short Store Local Word. Store (ETOS) into word with offset x in the current activation record.
STLB	UB	127	Store Local Word/Byte Offset. Store (ETOS) into word with offset UB in the current activation record.
STLW	W	128	Store Local Word/Word Offset. Store (ETOS) into word with offset W in the current activation record.

Implementation Note: The address of the first local (offset 0) is contained in the Local Pointer register (LP). The address of the Nth local is computed as (LP) + N.

4.A.1.c Own One Word Loads and Stores

LDO0..15		139-154	Short Load Own Word. LDOx fetches the word with offset x in the current Global Data Block (GDB) and pushes it on the EStack.
LDOB	UB	137	Load Own Word/Byte Offset. Fetches the word with offset UB in the current Global Data Block (GDB) and pushes it on the EStack.
LDOW	W	138	Load Own Word/Word Offset. Fetches the word with offset W in the current Global Data Block (GDB) and pushes it on the EStack.
LOAB	UB	155	Load Own Address/Byte Offset. Pushes a word pointer to the word with offset UB in the current Global Data Block (GDB) on EStack.
LOAW	W	156	Load Own Address/Word Offset. Pushes a word pointer to the word with offset W in BASE activation record on EStack.
STO0..7		159-166	Short Store Own Word. STOx stores (ETOS) into the word with offset x in the current Global Data Block (GDB).
STOB	UB	157	Store Own Word/Byte Offset. Stores (ETOS) into the word with offset UB in the current Global Data Block (GDB).
STOW	W	158	Store Own Word/Word Offset. Stores (ETOS) into the word with offset W in the current Global Data Block (GDB).

Implementation Note: The address of the first own (offset 0) is contained in the Global Pointer register (GP). The address of the Nth own is computed as (GP)+N.

4.A.1.d Global One Word Loads and Stores

LDGB	UB1,UB2	192	Load Global Word/Byte Offset. Loads the word with offset UB2 in the Global Data Block (GDB) for program segment UB1 onto EStack.
LDGW	UB,W	193	Load Global Word/Word Offset. Same as LDGB except a full word offset is used.
LGAB	UB1,UB2	194	Load Global Address/Byte Offset. Pushes a word pointer to the word with offset UB2 in the Global Data Block (GDB) for program segment UB1 onto EStack.
LGAW	UB,W	195	Load Global Address/Word Offset. Same as LGAB except a full word offset is used.
LGAWW	W1,W2	181	Load Global Address/Word Segment, Word Offset. Same as LGAB except a full word is used both for the segment number and the offset.
STGB	UB1,UB2	196	Store Global Word/Byte Offset. Stores (ETOS) in word with offset UB2 in the Global Data Block (GDB) for program segment UB1.
STGW	UB,W	197	Store Global Word/Word Offset. Same as STGB except a full word offset is used.
			Note: To achieve LDGW and STGW with full word segment numbers, use LGAWW with LDIND or STIND.

Implementation Note: Self-relative pointers to the Global Data Blocks (GDB) for each externally referenced segment are contained in the External Segment Table (XST), pointed to by the Global Pointer (GP). The address of the first global (offset 0) in the designated GDB is computed as GP - 2 * ISN, where ISN (Internal Segment Number) is the program segment number specified in the load or store instruction. The Nth global is addressed by the base address (computes as above) plus N.

4.A.1.e Intermediate One Word Loads and Stores

LDIB	UB1,UB2	215	Load Intermediate Word/Byte Offset. UB1 indicates the number of static links to traverse to find the activation record to use. UB2 is the offset within the activation record of the desired word. The datum is pushed on EStack.
LDIW	UB,W	216	Load Intermediate Word/Word Offset. Same as LDIB except a word offset is used.
LIAB	UB1,UB2	217	Load Intermediate Address/Byte Offset. A word pointer is pushed on EStack (determined as in LDIB).
LIAW	UB,W	218	Load Intermediate Address/Word Offset. A word pointer is pushed on EStack (determined as in LDIW).
STIB	UB1,UB2	219	Store Intermediate Word/Byte Offset. Stores (ETOS) in memory (address determined as in LDIB).
STIW	UB,W	220	Store Intermediate Word/Word Offset. Stores (ETOS) in memory (address determined as in LDIW).

Implementation Note: The Activation Pointer register (AP) contains the address of the current Activation Control Block (ACB). Within the ACB is the Static Link (SL) to the previous ACB. To compute the address of the first intermediate word of the desired level, traverse the Static Links to the correct ACB. Within the ACB is the Local Pointer (LP) for that activation record.

4.A.1.f Indirect One Word Loads and Stores

STIND	21	Store Indirect. (ETOS) is stored into the word pointed to by word pointer (ETOS-1).
LDIND	173	Load Indirect. Word pointed to by word pointer (ETOS) is pushed on EStack.

4.A.2 Loads and Stores of Multiple Words

4.A.2.a Double Word Loads and Stores (Reals and Pointers)

LDDC <block> 237 — Load Double Word Constant. <block> is a double word constant. Load the constant onto EStack.

LDDW 239 — Load Double Word. (ETOS) is a word pointer to a double word. The double word is pushed onto EStack.

STDW 183 — Store Double Word. (ETOS),(ETOS-1) is a double word and (ETOS-2) is a word pointer to a double word block of memory. The double word is popped from ESTACK into the double word pointed to by (ETOS-2).

4.A.2.b Multiple Word Loads and Stores (Sets)

LDMC UB,<block> 236 — Load Multiple Word Constant. UB is the number of words to load, and <block> is a block of UB words, in reverse word order. Load the block onto the MStack.

LDMW 238 — Load Multiple words. (ETOS-1) is a word pointer to the beginning of a block of (ETOS) words. Push the block onto the MStack.

STMW 182 — Store Multiple Words. The MStack contains a block of (ETOS) words, (ETOS-1) is a word pointer to a similar block. Transfer the block from MStack to the destination block.

4.A.3 Byte Arrays

Note: A byte pointer is loaded onto the stack with a LLA, LOA or LGA of the base address of the array followed by the computation of the offset.

LDB		23	Load Byte. Push the byte (after zeroing the high Byte) pointed to by byte pointer (ETOS),(ETOS-1) on EStack.
STB		24	Store Byte. Store the low byte of (ETOS) into the location specified by byte pointer (ETOS-1),(ETOS-2).
MVBB	UB	167	Move Bytes/Byte Counter. (ETOS),(ETOS-1) is a source byte pointer to a block of UB bytes, and (ETOS-2),(ETOS-3) is the destination byte pointer to a similar block. Transfer the source block to the destination block.
MVBW		168	Move Bytes/Word Counter. Same as MVBB except (ETOS-1), (ETOS-2) is the source byte pointer, (ETOS-3), (ETOS-4) is the destination byte pointer, and (ETOS) is the number of bytes to transfer.

4.A.4 Strings

LSA	UB,<chars>	19	Load String Address. UB is the length of the string constant <chars>. A string pointer is pushed on EStack (the virtual address of UB is pushed followed by a zero). UB is word aligned.
SAS		184	String Assign. (ETOS-1),(ETOS-2) is the source string pointer, and (ETOS-3),(ETOS-4) is the destination string pointer. (ETOS) is the declared length of the destination. The length of the source and destination are compared, and if the source string is longer than the destination, a run-time error occurs. Otherwise all bytes of source containing valid information are transferred to the destination string.
LDCH		25	Load Character. (ETOS),(ETOS-1) is a string pointer. (ETOS) is checked to insure that it lies within the dynamic length of the string. If so, the character pointed to by (ETOS),(ETOS-1) is pushed; otherwise, a run-time error occurs.
STCH		28	Store Character. (ETOS) is a character and (ETOS-1),(ETOS-2) is a string pointer. (ETOS-1) is checked to insure that it lies within the dynamic length of the string. If so, the character (ETOS) is stored in the string, at the position pointed to by (ETOS-1),(ETOS-2); otherwise, a run-time error occurs.

4.A.5 Record and Array Indexing and Assignment

MOVB	UB	169	Move Words/Byte Counter. (ETOS) is a word pointer to a block of UB words, and (ETOS-1) is a word pointer to a similar block. The block pointed to by (ETOS) is transferred to the block pointed to by (ETOS-1).
MOVW		170	Move Words/Word Counter. Same as MOVB except (ETOS-1) is the source pointer, (ETOS-2) is the destination pointer, and (ETOS) is the number of words to be transferred.
SIND0-7		173-180	Short Index and Load Word. SINDx indexes the word pointer (ETOS) by x words, and pushes the word pointed to by the result on ESTACK. (Note: SIND0 is synonymous to LDIND).
INDB	UB	171	Static Index and Load Word/Byte Index. Indexes the word pointer (ETOS) by UB words, and pushes the word pointed to by the result on ESTACK.
INDW	W	172	Static Index and Load Word/Word Index. Same as INDB except a full word index is used.
INCB	UB	232	Increment Field Pointer/Byte Index. The word pointer (ETOS) is indexed by UB words and the resultant pointer is pushed on ESTACK.
INCW	W	233	Increment Field Pointer/Word Index. Same as INCB except a full word index is used.

Note: INCB and INCW are equivalent to add UB or W to (ETOS).

IXAB	UB	221	Index Array/Byte Array Size. (ETOS) is an integer index, (ETOS-1) is a word pointer to the base of the array, and UB is the size (in words) of an array element. A word pointer to the first word of the indexed element is pushed on ESTACK.
IXAW		222	Index Array/Word Array Size. Same as IXAB except (ETOS-1) is the integer index, (ETOS-2) is the word pointer to the base of the array, and (ETOS) is the size (in words) of an array element. (Gen1A) full word is

used for the array element size.

IXA1..4		223-226	Index Array/Short Array Size. Same as IXAB except array element sizes are fixed at 1-4.
IXP	UB	214	Index Packed Array. (ETOS) is an integer index, and (ETOS-1) is a word pointer the base of the array. Bits 4-7 of UB contain the number of elements per word minus 1, and bits 0-3 contain the field width (in bits) minus 1. Compute and push a packed field pointer.
LDP		26	Load a Packed Field. Push the field described by the packed field pointer (ETOS),(ETOS-1) on ESTACK.
STP		27	Store into Packed Field. Store (ETOS) in the field described by the packed field pointer (ETOS-1),(ETOS-2).
ROTSHI	UB	20	Rotate/Shift. (ETOS-1) is the argument to be rotated or shifted, and (ETOS) is the distance to rotate or shift. If UB is 0 then a right rotate occurs, and if UB is 1 then a shift occurs. The direction of the shift is determined from (ETOS); If (ETOS) >= 0 then a left shift occurs; otherwise, a right shift. (ETOS) must be in the range from -15 to +15.

4.B Top of Stack Arithmetic and Comparisons

4.B.1 Logical

LAND	30	Logical Add. AND (ETOS) into (ETOS-1).
LOR	31	Logical Or. OR (ETOS) into (ETOS-1).
LNOT	32	Logical Not. Take one's complement of (ETOS).
EQUBOOL	33	Boolean =,
NEQBOOL	34	<>,
LEQBOOL	35	<=,
LESBOOL	36	<,
GEQBOOL	37	>=,
GTRBOOL	38	and > comparisons. Compare (ETOS-1) to (ETOS) and push true or false on ESTACK.

4.B.2 Integer

ABI	71	Absolute Value of Integer. Take absolute value of (ETOS). Result is undefined if (ETOS) is initially -32768.
ADI	72	Add Integers. Add (ETOS) and (ETOS-1).
NGI	73	Negate Integer. Take the twos complement of (ETOS).
SBI	74	Subtract Integers. Subtract (ETOS) from (ETOS-1).
MPI	75	Multiply Integers. Multiply (ETOS) and (ETOS-1). This instruction may cause overflow if the result is larger than 16 bits.
DVI	76	Divide Integers. Divide (ETOS-1) by (ETOS) and push quotient (as defined by Jensen and Wirth).
MODI	77	Modulo Integers. Divide (ETOS-1) by (ETOS) and push the remainder (as defined by Jensen and Wirth).
CHK	78	Check Against Subrange Bounds. Insure that (ETOS-1) <= (ETOS-2) <= (ETOS), leaving (ETOS-2) on top of the stack. If conditions are not met a run-time error occurs.
EQUI	39	Integer =,
NEQI	40	<>,
LEQI	41	<=,
LESI	42	<,
GEQI	43	>=,
GTRI	44	and > comparisons. Compare (ETOS-1) to (ETOS) and push true or false on ESTACK.

4.B.3 Real Operations

ROPS UB 250 Real Operations. Arithmetic operations on floating point (32 bit) values. In general, (ETOS) is the low-order word and (ETOS-1) is the high-order word of the value. When two floating point values are involved, (ETOS-2) is the low-order word of the second real and (ETOS-3) is the high-order word.

All over/underflows cause a run-time error. Division by zero (0) also causes a run-time error.

UB determines the operation according to the following table:

0 - The real (ETOS),(ETOS-1) is truncated (as defined by Jensen and Wirth), converted to an integer, and pushed onto EStack.

1 - The integer (ETOS) is converted to a floating-point number and pushed onto EStack.

2 - Add (ETOS),(ETOS-1) and (ETOS-2),(ETOS-3).

3 - Negate the real (ETOS),(ETOS-1).

4 - Subtract (ETOS),(ETOS-1) from (ETOS-2),(ETOS-3).

5 - Multiply (ETOS),(ETOS-1) and (ETOS-2),(ETOS-3).

6 - Divide (ETOS),(ETOS-1) by (ETOS-2),(ETOS-3).

7 - The real (ETOS),(ETOS-1) is rounded (as defined by Jensen and Wirth), truncated and converted to an integer, and pushed onto EStack.

8 - Take the absolute value of the real (ETOS),(ETOS-1).

9 - = of two real values. (ETOS) = true or false.

10 - <> of two real values.

11 - <= of two real values.

12 - < of two real values.

13 - >= of two real values.

14 - > of two real values.

4.B.4 Sets

ADJ	UB	185	Adjust Set. The set on the top of the MSTACK is forced to occupy UB words, either by expansion or compression, and its length word is popped from ESTACK.
SGS		66	Build Singleton Set. The integer (ETOS) is checked to insure that 0 <= (ETOS) <= 4,095, the set [(ETOS)] is pushed on MSTACK, and the size of the set is pushed on ESTACK. If (ETOS) is out of range, the null set is pushed (a zero is pushed on ESTACK, the MSTACK is not altered).
SRS		68	Build SubRange Set. The integers (ETOS) and (ETOS-1) are checked as in SGS, the set [(ETOS-1)..(ETOS)] is pushed onto MSTACK, and the size of the set is pushed on ESTACK. (The null set is pushed if (ETOS-1) > (ETOS) or either is out of range).
INN		88	Set Membership. See if integer (ETOS) is in set contained on the top of MSTACK, and with length (ETOS-1), pushing TRUE or FALSE on ESTACK.
UNI		89	Set Union. The union of the two sets contained on the top of MSTACK (with sizes (ETOS) and (ETOS-1)) is pushed on MSTACK, and the length of the result on ESTACK.
INT		90	Set Intersection. The intersection of the two sets contained on the top of MSTACK (with sizes (ETOS) and (ETOS-1)) is pushed on MSTACK, and the length of the result on ESTACK.
DIF		91	Set Difference. The difference of the two sets contained on the top of MSTACK, and sizes (ETOS) and (ETOS-1) is pushed on MSTACK, and the length of the result on ESTACK.
EQUPOWR		63	Set =,
NEQPOWR		64	<>,

LEQPOWR	65	<= (subset of),
GEQPOWR	67	and >= (superset of)

comparisons of the two sets on top of ESTACK, with sizes (ETOS) and (ETOS-1).

4.B.5 Strings

EQUSTR	51	String =,
NEQSTR	52	<>,
LEQSTR	53	<=,
LESSTR	54	<,
GEQSTR	55	>=,
GTRSTR	56	and > comparisons. The string pointed to by string pointer (ETOS-2),(ETOS-3) is lexicographically compared to the string pointed to by string pointer (ETOS),(ETOS-1).

4.B.6 Byte Arrays

EQUBYT	UB	57	Byte Array =,
NEQBYT	UB	58	<>,
LEQBYT	UB	59	<=,
LESBYT	UB	60	<,
GEQBYT	UB	61	>=,
GTRBYT	UB	62	and >

comparisons. <=, <, >=, and > are only emitted for packed arrays of characters. The argument, UB, if non-zero, is the size of the array. If UB is equal to 0, then (ETOS) is the size of the array.

4.B.7 Array and Record Comparisons

EQUWORD UB 69 Word or multiword structure =

NEQWORD UB 70 and <>

comparisons. The argument, UB, if non-zero, is the size of the array. If UB equals 0, then (ETOS) is the size of the array.

4.B.8 Long Operations

LOPS UB 252 Long Operations. Arithmetic operations on long (32 bit) values. In general, (ETOS) is the low-order word and (ETOS-1) is the high-order word of the value. When two long values are involved, (ETOS-2) is the low-order word of the second long and (ETOS-3) is the high-order word. UB determines the operation according to the following table:

0 - Converts the long value (ETOS), (ETOS-1) to a single word. The high- order word must be 0 or all 1's, as it is truncated. If not, a runtime error is generated.

1 - Converts a single word (ETOS) into a long value.

2 - Adds two long values.

3 - Negates long value.

4 - Subtracts two long values.

5 - Multiplies two long values.

6 - Divides two long values.

7 - Mods two long values.

8 - Absolute value of a long value.

9 - = of two long values. (ETOS) = true or false.

10 - <> of two long values.

11 - <= of two long values.

12 - < of two long values.

13 - >= of two long values.

14 - > of two long values.

4.C Jumps

JMPB	B	204	Unconditional Jump/Byte Offset. B is added to the IPC. Negative values of B cause backward jumps.
JMPW	W	205	Unconditional Jump/Word Offset. W is added to the IPC. Negative values of W cause backward jumps.
JFB	B	206	False Jump/Byte Offset. Jump (as in JMPB) if (ETOS) is false.
JFW	W	207	False Jump/Word Offset. Jump (as in JMPW) if (ETOS) is false.
JTB	B	208	True Jump/Byte Offset. Jump (as in JMPB) if (ETOS) is true.
JTW	W	209	True Jump/Word Offset. Jump (as in JMPW) if (ETOS) is true.
JEQB	B	210	Equal Jump/Byte Offset. Jump (as in JMPB) if integer (ETOS) equals (ETOS-1).
JEQW	W	211	Equal Jump/Word Offset. Jump (as in JMPW) if integer (ETOS) equals (ETOS-1).
JNEB	B	212	Not Equal Jump/Byte Offset. Jump (as in JMPB) if integer (ETOS) is not equal to (ETOS-1).
JNEW	W	213	Not Equal Jump/Word Offset. Jump (as in JMPW) if integer (ETOS) is not equal to (ETOS-1).

XJP W1,W2,W3,<Case Table> 100

Case Jump. W1 is word-aligned, and is the minimum index of the table. W2 is the maximum index. W3 is the offset to the code to be executed if the case specified has no entry in the case table. The case table is W2 - W1 + 1 words long and contains offsets to the code to be executed for each case.

If (ETOS), the actual index, is not in the range W1..W2 then W3 is added to PC. Otherwise, (ETOS) - W1 is used as an index into the case table and the index entry is

added to PC.

4.D Routine Calls and Returns

Note: There can be at most 256 routines in a segment.

CALL	UB	186	Call Routine. Call routine UB, which is in the current segment.
CALLXB	UB1,UB2	234	Call External Routine/Byte Segment. UB1 is the internal segment number (ISN) which contains the routine numbered UB2 to be called. First the ISN is translated to the correct SSN, and residency of that segment is checked. If the segment is resident, the call proceeds; if not, the PC is backed up so that the call will be re-executed, and a segment fault occurs. The second attempt is guaranteed to succeed, since the process is unable to resume execution until the segment SSN is resident.
CALLXW	W,UB	235	Call External Routine/Word Segment. Same as CALLXB except the internal segment number (ISN) is given in a full word.
LVRD	W,UB1,UB2	98	Load Variable Routine Descriptor. This Q-Code pushes a Variable Routine Descriptor on the EStack for the routine UB1 in segment ISN W, at lexical level UB2. The following values (which comprise a variable routine descriptor) are pushed: (ETOS) = System Segment Number (SSN); (ETOS-1) = Global Pointer, represented as an offset from SB; (ETOS-2) = Routine Number; and (ETOS-3) = Static Link (determined as if a call were actually performed to the routine here).
CALLV		187	Call Variable Routine. The ESTACK elements (ETOS) --- (ETOS-3) are a variable routine descriptor (as described above in LVRD). Residency of the segment are checked. If the segment is resident, the call is made as will CALL, except the GP and SL are taken from the variable routine descriptor; if not, a segment fault occurs as with CALLX.
RETURN		200	Return from Routine. Return from the current routine. If the routine was a function, the function value is left on the top of the MStack. Since the first word of a code segment is not code, but an offset to

the routine dictionary, if the RA which is being returned to is 0, the return is performed to the exit code of that routine. (This proves useful for the EXIT and EXGO Q-Codes described below).

EXIT W,UB 92 Exit from Routine. Exit from all routines up to and including the most recent invocation of the routine UB in ISN W. This is accomplished by setting the RAs in all the ACBs to 0, from the most recent through and including the first ACB which was created from an invocation the routine to be exited, and jumping to the exit code of the current routine.

EXGO W1,UB,W2 29 Exit and Goto. Exit from all routines up to, but not including, routine UB in ISN W1, and then jump to the instruction with offset W2 from CB. The implementation is similar to EXIT, except the last RA modified is loaded with W2.

ENABLE W,UB1,UB2 242 Enable Exception Handler. W and UB1 are the internal segment and routine numbers, respectively, of the exception being enabled. UB2 is the routine number of the handler. A new exception enable record is pushed (quad word aligned) onto the MStack and linked into the routine's current exception list.

RAISE W1,UB,W2 243 Raise Exception. W1 and UB1 are the internal segment and routine numbers, respectively, of the exception to be raised. W2 is the number of words of parameters that have already been pushed onto the memory stack. The exception is raised.

4.E Systems Programs Support Procedures

BREAK	254	Breakpoint QCode. Causes a Qcode level breakpoint to the microcode kernel (KRNL).
NOOP	93	No-Operation.
REPL	94	Replicate. Replicate (ETOS).
REPL2	95	Replicate Two. Replicate two top-of-estack words (i.e., first push original (ETOS-1), then push original (ETOS)).
MMS	96	Move to Memory Stack. Push (ETOS) onto MTOS (16-bit transfer).
MES	97	Move to Expression Stack. Push (MTOS) onto ETOS (16-bit transfer - top 4 bits are zeroed).
MMS2	201	Move Double to Memory Stack. Transfer the top two words from the EStack to the MStack. The order is reversed; old (ETOS) is (MTOS-1), (ETOS-1) is (MTOS).
MES2	202	Move Double to Expression Stack. Transfer the top two words from the MStack to the EStack. THe order is reversed; old (MTOS) is (ETOS-1), (MTOS-1) is (ETOS).
RASTER-OP	102	RasterOp. RasterOp is a special QCode which is used to manipulate blocks of memory of arbitrary sizes. It is especially useful for creating and modifying displays on the screen. RasterOp modifies a rectangular area (called the "destination") of arbitrary size (to the bit). The picture drawn into this rectangle is computed as a function of the previous contents of the destination and the contents of another rectangle of the same size called the "source". The functions performed to combine the two pictures are described below. RasterOp can be used on memory other than that used for the screen bitmap. There are two parameters that specify the areas of memory to be used for the source and destination: a pointer to the start of the memory block and the length (in words) of

scanlines in the block. A scanline is one of the elements that cross the block. On the screen, for example, a scanline is one of the horizontal lines with a length of 48 words. Within these regions, the positions of the source and destination rectangles are given as offsets from the pointer. Thus position (0,0) would be at the upper left corner of the region, and, for the screen, (767, 1023) would be the lower right.

The EStack must be arranged in the following order for RASTER-OP:

```
(ETOS-10) Function
(ETOS-9)  Width
(ETOS-8)  Height
(ETOS-7)  Destination-X-Position
(ETOS-6)  Destination-Y-Position
(ETOS-5)  Destination-Area-Line-Length
(ETOS-4)  Destination-Memory-Pointer
(ETOS-3)  Source-X-Position
(ETOS-2)  Source-Y-Position
(ETOS-1)  Source-Area-Line-Length
(ETOS)    Source-Memory-Pointer
```

The values on the stack are defined below:

"Function" defines how the source and the destination are to be combined to create the final picture stored at the destination. The RasterOp functions are as follows (Src represents the source and Dst the destination):

Function	Name	Action
0	RRpl	Dst gets Src
1	RNot	Dst gets NOT Src
2	RAnd	Dst gets Dst AND Src
3	RAndNot	Dst gets Dst AND NOT Src
4	ROr	Dst gets Dst OR Src
5	ROrNot	Dst gets Dst OR NOT Src
6	RXor	Dst gets Dst XOR Src
7	RXNor	Dst gets Dst XNOR Src

"Width" specifies the size in the horizontal ("x") direction of the source and destination rectangles (given in bits).

"Height" specifies the size in the vertical ("y") direction of the source and destination rectangles (given in scan lines).

"Destination-X-Position" is the bit offset of the left side of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Y-Position" is the scan-line offset of the top of the destination rectangle. The value is offset from Destination-Memory-Pointer (see below).

"Destination-Area-Line-Length" is the number of words which comprise a line in the destination region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Destination-Memory-Pointer" is the virtual address of the top left corner of the destination region. This pointer MUST be quad-word aligned, however.

"Source-X-Position" is the bit offset of the left side of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Y-Position" is the scan-line offset of the top of the source rectangle. The value is offset from Source-Memory-Pointer (see below).

"Source-Area-Line-Length" is the number of words which comprise a line in the source region (hence defining the region's width). The appropriate value to use when operating on the screen is 48. The specified value must be a multiple of four (4) and within the range 4 through 48.

"Source-Memory-Pointer" is the virtual address of the top left corner of the source region. This pointer MUST be quad-word aligned, however.

LINE	241	Line Drawing. (ETOS) is a pointer to the origin (relative 0,0) of the area on which the line is drawn. (ETOS-4) and (ETOS-3) are the x and y coordinates (respectively) of the first endpoint of the line. (ETOS-2) and (ETOS-1) are the x and y coordinates (respectively) of the second endpoints on the line. (ETOS-5) is the style of the line where a value of 1 means erase the line, 2 means to xor the line and anything else means to draw the line.
STARTIO	103	(ETOS) is the channel on which to start IO.
INTOFF	105	Disable interrupts.
INTON	106	Enable interrupts.
EXCH	230	Exchange. (ETOS) and (ETOS-1) are swapped.
EXCH2	231	Exchange Double. The pair (ETOS) and (ETOS-1) are swapped with the pair (ETOS-2) and (ETOS-3).
TLATE1	227	Translate Top of Stack. (ETOS),(ETOS-1) is a virtual address. If the segment SSN (ETOS-1) is resident, convert the virtual address to an offset from stack base (SB) and execute the next Q-Code (what ever it may be), with out interrupts, to competion. If the segment SSN (ETOS-1) is non-resident, restore the EStack to its previous state, backup the PC to re-execute the TLATE1 and perform a segment fault.
TLATE2	228	Translate Top of Stack - 1. Same as TLATE1 except the virtual address is at (ETOS-1),(ETOS-2).
TLATE3	229	Translate Top of Stack - 2. Same as TLATE1 except the virtual address is at (ETOS-2),(ETOS-3).

STLATE	UB	240	Special Translate. This translate is similar to the previous translate Q-Codes, except that it can specify a greater depth than TLATE3, and that it may specify the translation of 2 virtual addresses. Each half of UB is interpretted as the depth of the System Segment Number word of the virtual address to be translated (prior to any stack alteration). A depth of 0 indicates no translation. All segments specified in the STLATE must be resident before any translations occur; otherwise a segment fault occurs. Note, if both nibbles of UB are non-zero then the low-order nibble (bits 0-3) must be less than the high-order nibble (bits 4-7).
LSSN		99	Load Stack Segment Number. Pushes the system segment number of the MStack onto EStack.
LDTP		203	Load Top Pointer (plus 1). Pushes the value of Top Pointer (TP) plus 1 onto EStack.
LDAP		244	Load Activation Pointer. The current activation pointer (as an offset from the base of the stack) is pushed onto the EStack.
ATPB	SB	188	Add to Top Pointer/Byte Value. Adds SB to TP.
ATPW		189	Add to Top Pointer/Word Value. Adds (ETOS) to TP.
WCS		190	Write Control Store. A control store word is written from information on the EStack. (ETOS) is the address (with bytes exchanged) in the control store to which the word will be written. (ETOS-1) is the value to be written into the high-order third, (ETOS-2) is the value to be written into the middle third and (ETOS-3) is to be written into the low-order third.
JCS		191	Jump to a Location in the Control Store. Control is transferred to the control store address (with bytes exchanged) given in (ETOS). A routine called with JCS should exit with a NextInst(0) jump.

REFILLOP	255	Refill the OpFile. This instruction causes execution to proceed from the beginning of the next quad-word.
INCDDS	251	Increment Diagnostic Display. The value of the diagnostic display is incremented and the contents of the EStack is checked. If the EStack is not empty, a runtime error is generated.

INDEX

How to Make a New System

John P. Strait

This manual describes how to change modules contained in the Three Rivers PERQ Operating System and how to create a new version of that system.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Table of Contents

The Scope of This Manual

This is a "how to" manual. If you follow the instructions of this manual, barring errors in the manual and bugs in your modifications to the system, you should be successful in making a new system. The instructions in this manual are not guaranteed to show you the most efficient way of making your changes. It only shows you a reliable way of making the changes.

This manual doesn't attempt to explain why the system is organized the way it is nor why each step is required. If your changes are not too major, this manual will help you to create a new system. It does not explain how to change low-level interface (e.g., interface to the Stream module) or low-level data structures (e.g., the segment tables or the structures on disk).

Recommendations

We make several recommendations to help you avoid errors that will make your system non-bootable. Remember, if you can no longer boot your PERQ, you can always boot from the "PERQ System Boot Floppy". Therefore, you always have a way of bringing up your PERQ.

1. Back up important files on floppy disks before you begin changing the PERQ Operating System. While it is improbable that you will destroy any files by changing the system, the importance of backup files cannot be stressed too much.

2. Maintain at least one partition of the disk that runs the old system. This enables you to boot your PERQ in the event that your new system contains bugs. When you receive your PERQ, the hard disk is divided into several partitions. At least one partition contains a pair of boot files: System.<n>.<x>.Boot and System.<n>.<x>.MBoot. The <n> in the file name is the current system version number, and the <x> in the file name represents the character that you hold down to use the corresponding boot files and partitions. The .Boot file contains the Operating System, and the .MBoot file contains the QCode interpreter microcode.

3. Do not change System.<n>.a.Boot and System.<n>.a.MBoot until your new system is completely debugged. These are the default boot files. If you do not change these files, you can boot the old system in the case that your new system contains errors.

4. Create a new directory or select an unused partition for your new experimental files when you begin making your changes to the system. Copy the sources of the files you want to change into this area before you begin editing. Compile all new .Seg files into this area. Do not use the root directory in the default partition: the one that is entered by the default boot letter ("a" is default--the same as not holding down a key). The default boot files are System.<n>.a.Boot and System.<n>.a.MBoot.

If you do not change the default boot files or the files in the root directory of the default partition, you will still have source and binary files that you can fall back on.

Overview

Creating a new system usually consists of the following steps.

1. Evaluate the change you intend to make
2. Create a directory to work in
3. Edit and compile system modules
4. Edit and compile system programs
5. Link the system and system programs
6. Prepare the system configuration file
7. Write a boot file
8. Test the new system
9. Iterate at step 3

Evaluate the Change You Intend to Make

Before you begin, you should determine how extensive the changes are. The following criteria tell you how much you need to change.

1. Are you changing the existing exports of any system modules? If not, you need only re-compile those modules that you change. If so, you may need to re-compile those modules and programs that import the ones you are changing.

2. Are you adding exports but not changing any that already exist? If you don't change existing exports, you need to re-compile only those modules that you change. You must, however, add your new exports at the end of the export list. By adding at the end of the export list, you do not change the storage allocation of existing variables or the routine numbers of existing procedures and functions. If you change either of these, you must re-compile all modules and programs that import the ones you are changing.

3. Are you changing the definition of data structures which are known by the microcode (e.g., memory manager tables) or data structures that live across boots (e.g., structures on disk)? If so, you may need to do a complicated bootstrapping operation to bring up your new system. This is beyond the scope of this manual.

4. Are you changing the existing exports of modules that the compiler knows about (Code, Dynamic, and Stream). If so, you again need to do a complicated bootstrapping operation. This too, is beyond the scope of this manual.

5. Are you changing the format of .Seg files? If so, you need to do a complicated bootstrapping operation which is beyond the scope of this manual.

Create a Directory, Edit, and Compile

Create a new directory for your experimental files. You do this with the MakeDir utility program. This new directory should be in the partition which contains the old system (probably the Boot partition). Copy sources of the system modules and programs into this directory. You may choose instead to work in a partition which, up until now, has not been used. Using a new partition is somewhat safer than merely creating a new directory in some old partition.

Edit the modules and programs that you need to change. Re-compile those modules and programs that you have changed and any others indicated by your evaluation of your changes.

Link the New System

Once all necessary changes and compilations have been done, you should link the new system and system programs. Choose a new system version number. Three Rivers Computer Corporation intends to use the version numbers between 1 and 99 for releases of the official PERQ Operating System. You should avoid these numbers to prevent conflicts with future Three Rivers Computer's releases. For example, you should choose version number 100 for your new system.

The new run files for System, Login, Shell, and Link should be in the root directory of the partition which contains your new system. You should use the following link commands to link your new system (assuming that the partition name is Part):

```
Link SystemSystem.100/System
Link LoginLogin.100
Link ShellShell.100
Link LinkLink.100
```

Prepare the System Configuration File

Before you write the boot file, you must create a system configuration file which describes the swappability of segments in the system. You probably can just copy the configuration file for the current version of the operating system. The default configuration file is named System.<n>.Config. If you need to change the swappability of segments in the system, you can copy the old file and edit it. Each line in the file describes the swappability of a single segment in the form:

<segment name> <swappability>

The <swappability> is chosen from the following:

SW - segment is swappable.
LS - segment is swappable but the memory manager should be reluctant to swap it out (this is not implemented yet).
US - segment is not swappable but may be moved in memory.
UM - segment is neither swappable nor movable.

Names with asterisks are recognized as special segment names. They are chosen from the following list:

SAT - Segment address table (default UM)
SIT - Segment information table (default US)
Cursor - Display cursor (default UM)
Screen - Display screen (default UM)
Font - Character set (default US)
Stack - Run-time stack (default US)
Names - System segment names (default SW)
IO - Input/output tables (default UM)

The default for code segments (modules) is US. We strongly suggest that you do not change the swappability of the special segments and, unless you are sure you know what you are doing, do not change the swappability of existing system modules. System data segments that the hardware or microcode uses cannot be moved, most data used by the operating system cannot be swapped, and the code that makes up the swapping system itself cannot be swapped. Since the default for code segments is US, you should add entries to the configuration file if you add modules to the system.

The default system configuration file is:

```
*SAT* UM
*SIT* US
*Cursor* UM
*Screen* UM
*Font* US
*IO* UM
System SW
Stream SW
Writer SW
IOErrMessages SW
Loader SW
```

```
Reader SW
Perq_String SW
Screen SW
FileSystem SW
GetTimeStamp SW
FileDefs SW
Memory SW
IO_Init SW
RunRead SW
FileDir SW
Scrounge SW
```

Write a New Boot File

You are now ready to write a boot file using the MakeBoot program. Before you run MakeBoot, choose a boot-letter for this new system; use one not already in use. You can run the Details program to find out which letters are in use. After choosing a boot letter, run MakeBoot and answer the questions in the following way:

-- Underlined text is what the PERQ types
-- Commentary is given inside { }
-- <CR> means type the RETURN key without entering any text
-- In this example, assume you have chosen the boot letter "z"

:MakeBoot

Root file name: System.100

Configuration file name [System.100.Config]: <CR>

Which character to boot from? z

Do you want to write the boot area [No]: <CR>

{ The boot area of the disk contains a microprogram which runs diagnostics and reads the .Boot and .MBoot files. You need to rewrite this only if you are making modifications to Vfy.Micro or SysB.Micro. }

Write a system boot file [Yes]: <CR>
Enter name of new system boot file [System.100.z.Boot]: <CR>
Existing boot file to copy (type return to build a new one): <CR>
Enter name of character set [Fix13.Kst]: <CR>

{ This writes the boot file containing the Pascal part of the system and special system segments such as the segment tables, the cursor, and the character set. Note that you may specify a character set which is different than the standard (Fix13.Kst). If you use a non-standard character set, some programs (like the Editor) may not work well. }

```
Write an interpreter boot file [Yes]: <CR>
Enter name of new micro boot file [System.100.z.MBoot]: <CR>
Existing boot file to copy (type return to build a new one): <CR>
Use standard interpreter microcode files? [Yes]: <CR>
Interpreter microcode file: ETHER10
Interpreter microcode$file: <CR>
```

{ This writes the boot file containing the microcode which is the Q-machine interpreter. Unless you are changing the interpreter microcode, you need only write this part once for a given boot letter. Note that you may add other microcode files to the boot file (as long as they do not overlap the standard microcode). }

Test the New System

You are now ready to boot your new system and test it. Hold down the boot key you selected ("z" in the example) and press the Boot button. If all goes well, your new system will announce itself. Note that when you try to run most programs, the loader informs you that they were linked under the old system. This means you must re-link them for your new system. It is a good idea to create another new directory to hold these run files. By putting the new run files in a directory by themselves, these run files will not get in your way when you are running the old system. If you want, you can make a Login profile to add this directory to your search list when you log in under your new system.

Once you are running the new system, you need to link the system utility programs. Set your path to the new directory that you created to contain the run files. Push the directory containing the old .Seg files onto your search list, and then push the directory containing the new .Seg files. Now, type:

```
Link ProgramName
```

for each utility program you wish to link.

Re-compile any programs that import modules whose exports have changed.

If your system doesn't come up at all, you can look at the diagnostic display to determine where in system initialization the system hangs.

Rewrite Other Boot Files

Once your new system is debugged and working, you can use MakeBoot to rewrite the boot files associated with other boot letters. Before you rewrite the old boot files, you must be sure that some partition contains all files that make up the new system. This includes files that you have not changed. If you fail to make a partition containing all source, binary, and run files, you run the risk of deleting portions of your new system when you delete the old system.

Perq.Files - PERQ Files Information

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Perq.Files - PERQ Files Information

Perq.Files is a list of the files distributed with the Three Rivers Computer Corporation PERQ. Source files are included in this list, but please note that they are included with the PERQ only if a source license is purchased. The User Library programs are also listed, but must be ordered from Trust - The Three Rivers Users' Society.

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

>OS.SYSTEM.SOURCE - OPERATING SYSTEM SYSTEM SOURCES

file name	version file name on floppy short description.
ACB.DFS	1.1 ACB.DFS Activation control block definitions--common to microcode and Pascal.
ALIGNMEMORY.PAS	1.1 ALIGNM.PAS Allocate buffers on multiples of 256 word boundaries.
ARITH.PAS	2.2 ARITH.PAS Double precision arithmetic for the disk system.
CODE.PAS	1.6 CODE.PAS Run file and seg file definitions.
CONTROLSTORE.PAS	1.2 CONTRO.PAS Load controlstore and jump to controlstore.
DYNAMIC.PAS	1.4 DYNAMI.PAS Dynamic allocation routines - New and Dispose.
EEB.DFS	1.0 EEB.DFS Exception Enable Block definitions--common to Pascal and microcode.
EXCEPT.PAS	2.9 EXCEPT.PAS The exceptions module.

EXCEPT.DFS	1.4	EXCEPT.DFS
	Definitions of the exceptions--common to microcode and Pascal.	
GETTIMESTAMP.PAS	1.4	GETTIM.PAS
	Get time and date as TimeStamp.	
LIGHTS.PAS	1.2	LIGHTS.PAS
	Definitions of the lights.	
LOADER.PAS	2.6	LOADER.PAS
	Perq Q-Code loader.	
MEMORY.PAS	2.13	MEMORY.PAS
	Memory manager.	
MOVEMEM.PAS	1.6a	MOVEME.PAS
	Memory manager utility to move segments.	
PASLONG.PAS	0.0	PASLON.PAS
	Handles double-precision integers.	
PASREAL.PAS	0.1	PASREA.PAS
	Real numbers.	
PERQ_STRING.PAS	2.4	PERQST.PAS
	String manipulation package.	
RD.DFS	1.1	RD.DFS
	Routine dictionary definitions--common to microcode and Pascal.	
READER.PAS	2.1	READER.PAS
	Stream package input conversion routines.	
REALFUNCTIONS.PAS	1.0	REALFU.PAS
	Standard floating-point functions.	
RUNREAD.PAS	1.1	RUNREA.PAS
	Procedures to read run files.	
RUNWRITE.PAS	1.1	RUNWRI.PAS
	Procedures to write run files.	
SCROUNGE.PAS	0.14	SCROUN.PAS
	The preliminary debugger.	
STREAM.PAS	1.20	STREAM.PAS
	Stream package base routines - Get and Put.	
SYSTEM.PAS	2.4	SYSTEM.PAS
	Operating system main program.	

SYSTEMDEFS.PAS	1.2	SYSDEF.PAS
	Common system definitions.	
VIRTUAL.PAS	2.8	VIRTUA.PAS
	The Swapper.	
VRD.DFS	1.0	VRD.DFS
	Variable Routine Descriptor Definitions common to Pascal and microcode.	
WRITER.PAS	2.2	WRITER.PAS
	Stream package output conversion routines.	

>OS.SYSTEM.BINARY - OPERATING SYSTEM SYSTEM SEGMENT FILES

file name	version	file name on floppy
	short description.	
ARITH.SEG	2.2	ARITH.SEG
ALIGNMEMORY.SEG	1.1	ALIGNM.SEG
CONTROLSTORE.SEG	1.2	CONTRO.SEG
CODE.SEG	1.6	CODE.SEG
DYNAMIC.SEG	1.4	DYNAMI.SEG
EXCEPT.SEG	2.9	EXCEPT.SEG
GETTIMESTAMP.SEG	1.4	GETTIM.SEG
LOADER.SEG	2.6	LOADER.SEG
MEMORY.SEG	2.13	MEMORY.SEG
MOVEMEM.SEG	1.6a	MOVEME.SEG
PASREAL.SEG	-	PASREA.SEG
PASLONG.SEG	-	PASLON.SEG
PERQ_STRING.SEG	2.4	PERQST.SEG
READER.SEG	2.1	READER.SEG
REALFUNCTIONS.SEG	1.0	REALFU.SEG
RUNREAD.SEG	1.1	RUNREA.SEG
RUNWRITE.SEG	1.1	RUNWRI.SEG
SCROUNGE.SEG	0.14	SCROUN.SEG
STREAM.SEG	1.20	STREAM.SEG
:LINK	-	SYSTEM.6=SYSTEM/SYSTEM
SYSTEM.SEG	2.4	SYSTEM.SEG
VIRTUAL.SEG	2.8	VIRTUA.SEG
WRITER.SEG	2.2	WRITER.SEG

>OS.PROGRAMS.SOURCE - OPERATING SYSTEM PROGRAM SOURCES

file name	version	file name on floppy
	short description.	
CLOCK.PAS	1.6	CLOCK.PAS
	Get, set, convert time and date as TimeStamp or String.	

CMDPARSE.PAS	3.6	CMDPAR.PAS Command parser.
DOSWAP.PAS	1.0	DOSWAP.PAS Module for handling the "swap" command for shell.
GPIB.PAS	1.3	GPIB.PAS Routines for dealing with the Perq IEEE-488 bus.
HELPER.PAS	1.1	HELPER.PAS Module for presenting help menu.
INITSHELL.PAS	2.2	INITSH.PAS Does initialization for Shell.
LINK.PAS	4.3	LINK.PAS Q-Code linker.
LOGIN.PAS	2.0	LOGIN.PAS Login program.
MULTIREAD.PAS	1.0	MULTIR.PAS Does a very fast multisector read of a file.
POPCMDPARSE.PAS	1.8	POPCMD.PAS Popup window command parse.
POPUP.PAS	2.4	POPUP.PAS Provides a Pop Up Menu facility for POS.
POPUPCURS.PAS	2.1	POPUPC.PAS Defines cursors for Pop Up.
PROFILE.PAS	1.1	PROFILE.PAS Module for accessing the profile.
QUICKSORT.PAS	1.2	QUICKS.PAS Sorts arrays, integers and strings.
RANDOMNUMBERS.PAS	1.2	RANDOM.PAS High-quality random number generator.
RS232BAUD.PAS	1.1	RS232B.PAS Set RS232 baud rate with optional enable input.
SHELL.PARAS	-	SHELL.PAR Information necessary to generate SHELL/HELP.
SHELL.PAS	3.4	SHELL.PAS Top-level command processor.

SHELLDEFS.PAS	1.0	SHELLD.PAS
	Definition of SHELL. Data format.	
USERPASS.PAS	1.3	USERPA.PAS
	Lookup user name/password pairs.	
UTILPROGRESS.PAS	1.16	UTILPR.PAS
	Module for showing progress of utility programs.	

>OS.PROGRAMS.BINARY - OPERATING SYSTEM PROGRAMS BINARY FILES

file name	version	file name on floppy
	short description.	
CLOCK.SEG	1.6	CLOCK.SEG
CMDPARSE.SEG	3.6	CMDPAR.SEG
DOSWAP.SEG	1.0	DOSWAP.SEG
GPIB.SEG	1.3	GPIB.SEG
HELPER.SEG	1.1	HELPER.SEG
:LINK	-	SHELL.6=SHELL
SHELL.SEG	3.4	SHELL.SEG
INITSHELL.SEG	2.2	INITSH.SEG
:LINK	-	LINK.6=LINK
LINK.SEG	4.3	LINK.SEG
:LINK	-	LOGIN.6=LOGIN
LOGIN.SEG	2.0	LOGIN.SEG
PROFILE.SEG	1.1	PROFIL.SEG
USERPASS.SEG	1.3	USERPA.SEG
MULTIREAD.SEG	1.0	MULTIR.SEG
POPCMDPARSE.SEG	1.8	POPCMD.SEG
POPUP.SEG	2.4	POPUP.SEG
POPUPCURS.SEG	2.1	POPUPC.SEG
QUICKSORT.SEG	1.2	QUICKS.SEG
RANDOMNUMBERS.SEG	1.2	RANDOM.SEG
RS232BAUD.SEG	1.1	RS232B.SEG
SHELLDEFS.SEG	1.0	SHELLD.SEG
UTILPROGRESS.SEG	1.16	UTILPR.SEG

>OS.IO.SOURCE - INPUT/OUTPUT SYSTEM SOURCE MODULES

file name	version	file name on floppy
	short description.	
ALLOCDISK.PAS	2.8	ALLOCD.PAS
	Allocation of sectors from the disk free list.	
DISKIO.PAS	3.12	DISKIO.PAS
	Medium-level disk input/output routines.	

ETHER10IO.PAS	1.8	ETHER1.PAS
	Ethernet IO interface.	
ETHERINTERRUPT.PAS	1.1	ETHERI.PAS
	Interrupt service for 10 MBaud ethernet.	
FILEACCESS.PAS	1.7	FILEAC.PAS
	File system segment routines.	
FILEDEFS.PAS	1.6	FILEDE.PAS
	Definitions used by the File System.	
FILEDIR.PAS	2.6	FILEDI.PAS
	File system directory routines.	
FILESYSTEM.PAS	7.3	FILESY.PAS
	File system--high-level disk input/output routines.	
FILETYPES.PAS	1.2	FILETY.PAS
	Definitions for the file type field.	
FILEUTILS.PAS	1.10	FILEUT.PAS
	File utilities not needed by system.	
IO.PAS	4.8	IO.PAS
	Input/output manager.	
IOERRORS.PAS	1.2	IOERRO.PAS
	Input/output error number constants.	
IOERRMESSAGES.PAS	1.1	IOERRM.PAS
	Names for the Input/output errors.	
IO_INIT.PAS	5.9	IOINIT.PAS
	Input/output manager initialization.	
IO_OTHERS.PAS	5.7	IOOTHE.PAS
	Other Input/output manager procedures and functions.	
IO_PRIVATE.PAS	5.9	IOPRIV.PAS
	Interrupt procedures and private definitions.	
IO_UNIT.PAS	6.2	IOUNIT.PAS
	The basic UnitIO procedures and functions.	
PMATCH.PAS	2.5	PMATCH.PAS
	Directory search with wild cards.	
RASTER.PAS	-	RASTER.PAS
	Raster-op definitions.	

READDISK.PAS	1.4	READDI.PAS
	Upper-level disk input/output routines.	
SCREEN.PAS	3.12	SCREEN.PAS
	Screen manager.	

>OS.IO.BINARY - OPERATING SYSTEM IO BINARY FILES

file name	version	file name on floppy
	short description.	
ALLOCDISK.SEG	2.8	ALLOCD.SEG
DISKIO.SEG	3.12	DISKIO.SEG
ETHER10IO.SEG	1.8	ETHER1.SEG
ETHERINTERRUPT.SEG	1.1	ETHERI.SEG
FILEACCESS.SEG	1.7	FILEAC.SEG
FILEDEFS.SEG	1.6	FILEDE.SEG
FILEDIR.SEG	2.6	FILEDI.SEG
FILESYSTEM.SEG	7.3	FILESY.SEG
FILEUTILS.SEG	1.10	FILEUT.SEG
IO.SEG	4.8	IO.SEG
IOERRMESSAGES.SEG	1.1	IOERRM.SEG
IO_INIT.SEG	5.9	IOINIT.SEG
IO_OTHERS.SEG	5.7	IOOTHE.SEG
IO_PRIVATE.SEG	5.9	IOPRIV.SEG
IO_UNIT.SEG	6.2	IOUNIT.SEG
PMATCH.SEG	2.5	PMATCH.SEG
READDISK.SEG	1.4	READDI.SEG
SCREEN.SEG	3.12	SCREEN.SEG

>OS.MISCELLANEOUS - OPERATING SYSTEM SPECIAL FILES

file name	version	file name on floppy
	short description.	
DEFAULT.PROFILE	-	DEFAUL.PRO
	Default login profile.	
DELETE.CURSOR	-	DELETE.CUR
	Cursor used when delete is busy.	

DIRTREE.CURSOR - DIRTRE.CUR
Cursor used by Dirtree program.

FIX13.KST - FIX13.KST
System character set file.

SCAVENGER.ANIMATE - SCAVEN.ANI
File of cursors for scavenger.

SYSTEM.USERS - SYSTEM.USE
Valid system users and passwords.

SYSTEM.6.CONFIG - SYSTEM.CON
Description of swappability for use by MakeBoot.

UTILPROGRESS.CURSOR - UTILPR.CUR
Cursor used by Utilprogress.

>OS.NOSOURCE.SOURCE - OPERATING SYSTEM NONSOURCE FILES

file name version file name on floppy
short description.

ACB.DFS 1.1 ACB.DFS
Activation control block definitions--common to microcode and Pascal.

ALLOCDISK.PAS 2.8 ALLOCD.PAS
Allocation of sectors from the disk free list.

ARITH.PAS 2.2 ARITH.PAS
Double precision arithmetic for the disk system.

CLOCK.PAS 1.6 CLOCK.PAS
Get, set, convert time and date as TimeStamp or String.

CMDPARSE.PAS 3.6 CMDPAR.PAS
Command parser.

CODE.PAS 1.6 CODE.PAS
Run file and seg file definitions.

CONTROLSTORE.PAS 1.2 CONTRO.PAS
Load controlstore and jump to controlstore.

DISKIO.PAS 3.12 DISKIO.PAS
Medium-level disk input/output routines.

DYNAMIC.PAS 1.4 DYNAMI.PAS
Dynamic allocation routines - New and Dispose.

EEB.DFS 1.0 EEB.DFS
Exception Enable Block definitions--common to Pascal and microcode.

ETHER10IO.PAS 1.8 ETHER1.PAS
Ethernet IO interface.

EXCEPT.PAS 2.9 EXCEPT.PAS
The exceptions module.

EXCEPT.DFS 1.4 EXCEPT.DFS
Definitions of the exceptions--common to microcode and Pascal.

FILEACCESS.PAS 1.7 FILEAC.PAS
File system segment routines.

FILEDEFS.PAS 1.6 FILEDE.PAS
Definitions used by the File System.

FILEDIR.PAS 2.6 FILEDI.PAS
File system directory routines.

FILESYSTEM.PAS 7.3 FILESY.PAS
File system--high-level disk input/output routines.

FILETYPES.PAS 1.2 FILETY.PAS
Definitions for the file type field.

FILEUTILS.PAS 1.10 FILEUT.PAS
File utilities not needed by system.

GETTIMESTAMP.PAS 1.4 GETTIM.PAS
Get time and date as TimeStamp.

GPIB.PAS 1.3 GPIB.PAS
Routines for dealing with the Perq IEEE-488 bus.

IO.PAS 4.8 IO.PAS
Input/output manager.

IOERRORS.PAS 1.2 IOERRO.PAS
Input/output error number constants.

IOERRMESSAGES.PAS 1.1 IOERRM.PAS
Names for the Input/output errors.

IO_INIT.PAS	5.9	IOINIT.PAS Input/output manager initialization.
IO_OTHERS.PAS	5.7	IOOTHE.PAS Other Input/output manager procedures and functions.
IO_PRIVATE.PAS	5.9	IOPRIV.PAS Interrupt procedures and private definitions.
IO_UNIT.PAS	6.2	IOUNIT.PAS The basic UnitIO procedures and functions.
LIGHTS.PAS	1.2	LIGHTS.PAS Definitions of the lights.
LOADER.PAS	2.6	LOADER.PAS Perq Q-Code loader.
MEMORY.PAS	2.13	MEMORY.PAS Memory manager.
MOVEMEM.PAS	1.6a	MOVEME.PAS Memory manager utility to move segments.
MULTIREAD.PAS	1.0	MULTIR.PAS Does a very fast multisector read of a file.
PASLONG.PAS	0.0	PASLON.PAS Handles double-precision integers.
PASREAL.PAS	0.1	PASREA.PAS Real numbers.
PERQ_STRING.PAS	2.4	PERQST.PAS String manipulation package.
PMATCH.PAS	2.5	PMATCH.PAS Directory search with wild cards.
POPCMDPARSE.PAS	1.8	POPCMD.PAS Popup window command parse.
POPUP.PAS	2.4	POPUP.PAS Provides a Pop Up Menu facility for POS.
POPUPCURS.PAS	2.1	POPUPC.PAS Defines cursors for Pop Up.
PROFILE.PAS	1.1	PROFILE.PAS Module for accessing the profile.

RANDOMNUMBERS.PAS	1.2	RANDOM.PAS
	High-quality random number generator.	
RASTER.PAS	-	RASTER.PAS
	Raster-op definitions.	
RD.DFS	1.1	RD.DFS
	Routine dictionary definitions--common to microcode and Pascal.	
READDISK.PAS	1.4	READDI.PAS
	Upper-level disk input/output routines.	
READER.PAS	2.1	READER.PAS
	Stream package input conversion routines.	
REALFUNCTIONS.PAS	1.0	REALFU.PAS
	Standard floating-point functions.	
RS232BAUD.PAS	1.1	RS232B.PAS
	Set RS232 baud rate with optional enable input.	
RUNREAD.PAS	1.1	RUNREA.PAS
	Procedures to read run files.	
RUNWRITE.PAS	1.1	RUNWRI.PAS
	Procedures to write run files.	
SCREEN.PAS	3.12	SCREEN.PAS
	Screen manager.	
SCROUNGE.PAS	0.14	SCROUN.PAS
	The preliminary debugger.	
STREAM.PAS	1.20	STREAM.PAS
	Stream package base routines - Get and Put.	
SYSTEM.PAS	2.4	SYSTEM.PAS
	Operating system main program.	
SYSTEMDEFS.PAS	1.2	SYSDEF.PAS
	Common system definitions.	
USERPASS.PAS	1.3	USERPA.PAS
	Lookup user name/password pairs.	
UTILPROGRESS.PAS	1.16	UTILPR.PAS
	Module for showing progress of utility programs.	
VIRTUAL.PAS	2.8	VIRTUA.PAS
	The Swapper.	

VRD.DFS	1.0 VRD.DFS Variable Routine Descriptor Definitions common to Pascal and microcode.
WRITER.PAS	2.2 WRITER.PAS Stream package output conversion routines.
>CANON.SOURCE	- CANON PRINTER SOURCE FILES
file name	version file name on floppy short description.
CANON.PAS	1.0 CANON.PAS Main PASCAL interface to CANON LBP-10 laser printer.
CAN.PAS	1.0 CAN.PAS CANON laser printer, fixed width font document print program.
CANP.PAS	1.0 CANP.PAS CANON laser printer, proportional spaced font document print program.
CANON.MICRO	1.0 CANON.MIC Microcode support required for, and loaded by CANON.PAS.
CANONBLOCK.PAS	1.0 CANONB.PAS PASCAL module to support memory bit image dump to CANON LBP-10.
CANONUTILS.PAS	1.0 CANONU.PAS General utility routines for CANON.
INFILE.PAS	1.0 INFILE.PAS High speed disk buffer reading support for CANON modules.
CBT.PAS	1.0 CBT.PAS Test program for CANONBLOCK.

>CANON.BINARY - CANON PRINTER BINARY FILES

file name	version	file name on floppy short description.
CAN40.KST	-	CAN40.KST
CAN40P.KST	-	CAN40P.KST
CANON.SEG	1.0	CANON.SEG
CAN.SEG	1.0	CAN.SEG
CANP.SEG	1.0	CANP.SEG
CANON.BIN	1.0	CANON.BIN
CANONBLOCK.SEG	1.0	CANONB.SEG
CANONUTILS.SEG	1.0	CANONU.SEG
INFILE.SEG	1.0	INFILE.SEG
CBT.SEG	1.0	CBT.SEG

>UTILITY.OTHERS.SOURCE - UTILITIES SOURCES

file name	version	file name on floppy short description.
BYE.PAS	2.2	BYE.PAS Logoff.
DETAILS.PAS	1.11	DETAIL.PAS Print system status information.
EDITOR.PAS	2.0	EDIT.PAS Editor main program.
EDITORI.PAS	2.0	EDITI.PAS Editor initialization module.
EDITORU.PAS	2.0	EDITU.PAS Editor utilities module.
EDITORK.PAS	2.0	EDITK.PAS Editor key-selection module.
EDITORT.PAS	2.0	EDITT.PAS Editor termination module.
EDITORK.PARAS	2.0	EDITK.PAR Souce to generate .HELP files for EDITOR.
EXPANDTABS.PAS	1.0	EXPAND.PAS Copy a text file and expand tabs to 8 character columns.
FINDSTRING.PAS	2.1	FINDST.PAS Searches files in directory for specified string.

GOODBY.MICRO	1.0	GOODBY.MIC
	Power down microcode.	
HELPGEN.PAS	0.0	HELPGE.PAS
	Process .PARAS files to make help files for use by Helper.	
MAKEBOOT.PAS	4.4	MAKEBO.PAS
	Make SYSTEM.nn.BOOT files.	
PATCH.PAS	1.7	PATCH.PAS
	Program to peek and poke into files.	
PERQ.FILES.PAS	1.3	PERQFI.PAS
	Program to gobble this file.	
SETTIME.PAS	2.0	SETTIM.PAS
	Sets the system date and time.	
USERCONTROL.PAS	1.3	USERCO.PAS
	Add/delete users from the password file.	

>UTILITY.OTHERS.BINARY - UTILITY OTHER BINARY FILES

file name	version	file name on floppy short description.
:LINK	2.2	BYE
BYE.SEG	-	BYE.SEG
GOODBY.BIN	-	GOODBY.BIN
.break :LINK		1.11 DETAILS
DETAILS.SEG	-	DETAIL.SEG
:LINK	2.0	EDITOR
EDITOR.SEG	-	EDIT.SEG
EDITORI.SEG	-	EDITI.SEG
EDITORK.SEG	-	EDITK.SEG
EDITORU.SEG	-	EDITU.SEG
EDITORT.SEG	-	EDITT.SEG
:LINK	1.0	EXPANDTABS
EXPANDTABS.SEG	-	EXPAND.SEG
:LINK		
FINDSTRING.SEG	2.1	FINDST.SEG
HELPGEN.SEG	0.0	HELPGE.SEG
:LINK	4.4	MAKEBOOT
MAKEBOOT.SEG	-	MAKEBO.SEG
:LINK	1.4	PATCH
PATCH.SEG	-	PATCH.SEG
:LINK	1.3	PERQ.FILES
PERQ.FILES.SEG	-	PERQFI.SEG
:LINK		
SETTIME.SEG	2.0	SETTIM.SEG
:LINK	1.3	USERCONTROL
USERCONTROL.SEG	-	USERCO.SEG

>UTILITY.DEVICE.SOURCE - UTILITIES DEVICES

file name	version file name on floppy short description.
CHATTER.PAS	0.6 CHATTE.PAS RS232 dumb terminal program.
FLOPPYCOPY.PAS	2.2 FLPCOP.PAS Floppydup part of floppy (formerly COPYFLOPPY).
FLOPPY.PAS	3.2 FLOPPY.PAS Main program for general floppy utility.
FLOPPYDEFS.PAS	0.0 FLPDEF.PAS Global defs for floppy.
FLOPPYUTILS.PAS	0.1 FLPUTI.PAS Utility routines for floppy.
FLOPPYFORMAT.PAS	0.1 FLPFOR.PAS Diskette format routines for floppy (formerly module FORMAT).
FTPUTILS.PAS	4.2 FTPUTILS.PAS Utilities for file transfer module.
FTPUSER.PAS	4.3 FTPUSE.PAS Implements all the commands.
FTP.PAS	4.3 FTP.PAS File transfer program.
PRINT.PAS	2.13 PRINT.PAS Print a text file on an HP 7310A printer or through the TNW GPIB to RS232 converter.
FLOPPYTRANSFERS.PAS	0.1 FLPTRA.PAS Utility routines for floppy (formerly RT11 and RT11Utils).
SETBAUD.PAS	0.0 SETBAU.PAS Program to set RS232 Baud rate.

>UTILITY.DEVICE.BINARY - UTILITY DEVICE BINARY FILES

file name	version	file name on floppy short description.
:LINK	0.6	CHATTER
CHATTER.SEG	-	CHATTE.SEG
:LINK	3.2	FLOPPY
FLOPPY.SEG	-	FLOPPY.SEG
FLOPPYDEFS.SEG	-	FLPDFS.SEG
FLOPPYCOPY.SEG	2.1	FLPCOP.SEG
FLOPPYUTILS.SEG	0.1	FLPUTI.SEG
FLOPPYFORMAT.SEG	0.1	FLPFOR.SEG
FLOPPYTRANSFERS.SEG	0.1	FLPTRA.SEG
:LINK	4.3	FTP.SEG
FTP.SEG	-	FTP.SEG
FTPUTILS.SEG	4.2	FTPUTI.SEG
FTPUSER.SEG	4.3	FTPUSE.SEG
:LINK	2.13	PRINT
PRINT.SEG	-	PRINT.SEG
:LINK	0.0	SETBAUD
SETBAUD.SEG	-	SETBAU.SEG

>UTILITY.FILE.SOURCE - FILE SYSTEM UTILITIES SOURCES

file name	version file name on floppy short description.
APPEND.PAS	3.1 APPEND.PAS Append a file to the end of another.
COPY.PAS	5.3 COPY.PAS Copy a file to another.
DELETE.PAS	2.4 DELETE.PAS Delete a file or files.
DIRECT.PAS	4.4 DIRECT.PAS Print a directory listing.
DIRTREE.PAS	3.2 DIRTRE.PAS Display the directory structure of a partition as a tree.
FIXPART.PAS	0.5 FIXPAR.PAS Fix smashed partition or disk information blocks.
MAKEDIR.PAS	2.2 MAKEDI.PAS Make directories.
PARTITION.PAS	3.2 PARTIT.PAS Initialize partitions on disks.
RENAME.PAS	5.3 RENAME.PAS Change the name of a file.
SCAVENGER.PAS	3.2 SCAVEN.PAS Analyze and reconstruct disks and directories.
DIRSCAVENGE.PAS	- DIRSCA.PAS Reconstruct directories.
SETSEARCH.PAS	1.3 SETSEA.PAS Change search lists.
TYPEFILE.PAS	4.2 TYPEFI.PAS Type file to the console.

>UTILITY.FILE.BINARY - UTILITY FILE SEG FILES

file name	version	file name on floppy short description.
:LINK	2.1	APPEND
APPEND.SEG	-	APPEND.SEG
:LINK	5.3	COPY
COPY.SEG	-	COPY.SEG
:LINK	2.4	DELETE
DELETE.SEG	-	DELETE.SEG
:LINK	4.4	DIRECT
DIRECT.SEG	-	DIRECT.SEG
:LINK	2.1	DIRTREE
DIRTREE.SEG	-	DIRTRE.SEG
:LINK	0.4	FIXPART
FIXPART.SEG	-	FIXPAR.SEG
:LINK	1.3	MAKEDIR
MAKEDIR.SEG	-	MAKEDI.SEG
:LINK	3.1	PARTITION
PARTITION.SEG	-	PARTIT.SEG
:LINK	5.3	RENAME
RENAME.SEG	-	RENAME.SEG
:LINK	3.2	SCAVENGER
SCAVENGER.SEG	-	SCAVEN.SEG
:LINK	1.2	SETSEARCH
SETSEARCH.SEG	-	SETSEA.SEG
:LINK	4.2	TYPEFILE
TYPEFILE.SEG	-	TYPEFI.SEG

>MICROCODE.SOURCE - MICROCODE SOURCES

file name	version file name on floppy short description.
PERQ.MICRO	2.4 PERQ.MIC Perq Q-code interpreter microcode.
PERQ.DFS	1.4 PERQ.DFS Definitions of registers, constants, and entrypoints used by Perq.Micro and other microprograms.
PERQ.QCODES.DFS	- QCODES.DFS Definitions of QCode instruction names and numbers.
PERQ.QCODE.1	- QCODE.1 Opcode interpreter routines for Perq.Micro (part 1).
PERQ.QCODE.2	- QCODE.2 Opcode interpreter routines for Perq.Micro (part 2).
PERQ.QCODE.3	- QCODE.3 Opcode interpreter routines for Perq.Micro (part 3).
PERQ.QCODE.4	- QCODE.4 Opcode interpreter routines for Perq.Micro (part 4).
PERQ.QCODE.5	- QCODE.5 Opcode interpreter routines for Perq.Micro (part 5).
PERQ.QCODE.6	- QCODE.6 Opcode interpreter routines for Perq.Micro (part 6).
PERQ.QCODE.7	- QCODE.7 Opcode interpreter routines for Perq.Micro (part 7).
PERQ.FLOAT.MUL	FLOAT.MUL Special multiply for floating point.
PERQ.ROUTINE.1	- ROUTIN.1 Subroutines for Perq.Micro (part 1).
PERQ.ROUTINE.2	- ROUTIN.2 Subroutines for Perq.Micro (part 2).

PERQ.INIT	- PERQ.INI Initialization for Perq.Micro.
RO.MICRO	0.6 RO.MIC Raster-op microcode.
LINE.MICRO	1.1 LINE.MIC Line drawing microcode.
IO.MICRO	1.10 IO.MIC Input/output microcode.
IO.DFS	1.5 IO.DFS Definitions of registers, constants, and entrypoints used by IO.Micro and other microprograms.
IOE3.MICRO	1.2 IOE3.MIC Microcode to drive the 3MBaud EtherNet.
VFY.MICRO	1.8 VFY.MIC Verify that the hardware seems to work.
SYSB.MICRO	2.5 SYSB.MIC System boot microcode.
BOOT.MICRO	4.0 BOOT.MIC Boot-prom microcode.
KRNL.MICRO	1.2 KRNL.MIC Perq microcode kernel.
LINK.MICRO	1.2 LINK.MIC 16-bit parallel interface microcode.

>MICROCODE.BINARY - MICROCODE BIN FILES

file name	version	file name on floppy short description.
PERQ.BIN	2.4	PERQ.BIN
IO.BIN	1.10	IO.BIN
VFY.BIN	1.8	VFY.BIN
SYSB.BIN	2.5	SYSB.BIN
BOOT.BIN	4.0	BOOT.BIN
KRNL.BIN	1.2	KRNL.BIN
LINK.BIN	1.2	LINK.BIN

>MICROCODE.MORE.SOURCE - MICROCODE SOURCES

file name	version	file name on floppy short description.
ETHER10.MICRO	4.0	ETHER1.MIC 10 MBaud ethernet microcode.

>MICROCODE.MORE.BINARY - MICROCODE MORE BINARY FILES

file name	version	file name on floppy short description.
ETHER10.BINARY	4.0	ETHER1.BIN

>MICROCODE.SUPPORT.SOURCE - MICROCODE SUPPORT SOURCES

file name	version	file name on floppy short description.
MICROOPTION.PAS	1.0	MICROO.PAS Option processor for PrqMic and PrqPlace.
ODTPRQ.PAS	8.0	ODTPRQ.PAS Simple Perq to Perq microcode debugger.
ODTUTILS.PAS	-	ODTUTI.PAS Utility routines for ODTPRQ.
ODTDUMP.PAS	-	ODTDUM.PAS The ODTPRQ dump subsystem.
PRQDIS.PAS	1.2	PRQDIS.PAS Perq microcode disassembler.
PRQPL_SORT.PAS	2.0	PRQPLS.PAS Sorting routines for PrqPlace.
PRQPLACE.PAS	2.5	PRQPLA.PAS Perq microcode placer.
PRQMIC.PAS	2.8	PRQMIC.PAS Perq microcode assembler.
PMEGEN.PAS	-	PMEGEN.PAS Program to create PRQMIC.ERROR from PRQMIC.ERR.TEXT.
PRQMIC.ERR.TEXT	-	PRQERR.TXT Source for PRQMIC.ERROR (error message text).

>MICROCODE.SUPPORT.BINARY - MICROCODE SUPPORT SEG FILES

file name	version	file name on floppy short description.
MICROOPTION.SEG	1.0	MICROO.SEG
PRQPL_SORT.SEG	2.0	PRQPLS.SEG
:LINK	2.8	PRQMIC
PRQMIC.SEG	-	PRQMIC.SEG
PRQMIC.ERROR	-	PRQMIC.ERR
:LINK	2.5	PRQPLACE
PRQPLACE.SEG	-	PRQPLA.SEG
:LINK	1.2	PRQDIS
PRQDIS.SEG	-	PRQDIS.SEG
:LINK	8.0	ODTPRQ
ODTPRQ.SEG	-	ODTPRQ.SEG
ODTUTILS.SEG	-	ODTUTI.SEG
ODTDUMP.SEG	-	ODTDUM.SEG
ODT13.KST	-	ODT13.KST

>DOCUMENTATION - DOCUMENTATION

file name	version file name on floppy short description.
EDITOR.DOC	- EDITOR.DOC Editor quick guide.
EDITORK.DOC	2.0 EDITK.DOC Editor User's Guide.
EXAMPLES.DOC	D.6 EXAMPL.DOC Programming examples.
FAULT.DOC	D.6 FAULT.DOC Fault dictionary for the diagnostic display.
FILE.FORMAT	- FILE.FOR Describe source file format for Perq software.
FILES.DOC	- FILES.DOC File system user's manual.
MICRO.DOC	D.6 MICRO.DOC Microprogrammer's guide.
MAKESYSTEM.DOC	- MAKESY.DOC How to make a new version of the operating system.
PASCAL.DOC	D.6 PASCAL.DOC Pascal extensions.
PERQ_Z80.DOC	- PERQZ8.DOC Description of PERQ/Z80 protocol.
QCODE.DOC	D.6 QCODE.DOC QCode reference manual.
SEGMENT.DOC	- SEGMEN.DOC Segment file format.
SETBAUD.DOC	- SETBAU.DOC User Information for program to set RS232 Baud rate.

>DOCUMENTATION.MORE - MORE DOCUMENTATION

file name	version file name on floppy short description.
UTILITIES.DOC	D.6 UTILIT.DOC Utility programs manual.
INTRO.DOC	D.6 INTRO.DOC Intorduction to the PERQ operating system.
PERQ.FILES	- PERQ.FIL This list.
PERQ.FILES.LABELS	- PERQFI.LAB Masters for labels on Perq.Files floppies.
POS.DOC	- POS.DOC Operating system interface guide.

>TEST.SOURCE - TEST PROGRAM SOURCES

file name	version file name on floppy short description.
CHARS.PAS	1.1 CHARS.PAS Screen magnifier.
CROSSHATCH.PAS	0.2 CROSSH.PAS Put crosshatch or checkerboard on display screen.
DISPATCH.MICRO	1.1 DISPAT.MIC Dispatch diagnostic.
DTST.MICRO	1.2 DTST.MIC Disk test microcode.
DUAL.MICRO	1.0 DUAL.MIC Microstore dual address test.
HIGH.MICRO	1.0 HIGH.MIC Test for stuck bits in the high bank of the microstore.
JUMP.MICRO	1.0 JUMP.MIC Test microcode jumps.
KEYTEST.PAS	1.5 KEYTST.PAS Keyboard test program.
LOOP.MICRO	0.0 LOOP.MIC Simple tests that repeat--allowing probing of boards.
LOW.MICRO	1.0 LOW.MIC Test for stuck bits in the low bank of the microstore.
MEM.MICRO	2.0 MEM.MIC Dual addressing test of the memory.
NEXTOP.MICRO	1.1 NEXTOP.MIC NextOp diagnostic.
	Register diagnostic.
NXTI.MICRO	1.1 NXTI.MIC NextInst diagnostic.

PART.MICRO	1.1	PART.MIC
	Memory Parity diagnostic.	
PBT.MICRO	1.1	PBT.MIC
	Pre-boot diagnostic.	
PDM.PAS	0.4	PDM.PAS
	Perq diagnostic monitor.	
PDM2.PAS	-	PDM2.PAS
	PDM 2nd module.	
PDCOMMON.PAS	-	PDCOMM.PAS
	PDM/PDS common definitions.	
PDMUTILS.PAS	-	PDMUTI.PAS
	PDM utility routines.	
PDMLOAD.PAS	-	PDMLOD.PAS
	PDM Pascal program loader.	
PDM.MAS	-	PDM.MAS
	PDM master file for current diagnostics.	
PDS.PAS	-	PDS.PAS
	Perq diagnostic slave.	
PDM.HELP	-	PDM.HELP
	PDM help file.	
PDM.MICRO	-	PDM.MIC
	Pseudo PDS microcode example.	
PDMVFY.MICRO	1.1	PDMVFY.MIC
	PDM version of VFY.	
RAT.MICRO	1.3	RAT.MIC
	Source data suspicious raster-op test.	
REGT.MICRO	1.1	REGT.MICRO
	Test of XY register file.	
SHIFT.MICRO	1.0	SHIFT.MIC
	Test of the shift hardware.	
STACK.MICRO	1.0	STACK.MIC
	20-bit, 16-level stack test.	
TESTFLOPPY.PAS	2.1	TSTFPY.PAS
	Test and format floppies.(formerly FLOPPY)	
TST.MICRO	1.1	TST.MIC
	Pre-Boot diagnostic.	

>TEST.BINARY - TEST PROGRAM SEG, RUN, AND BIN FILES

file name	version	file name on floppy short description.
:LINK	1.1	CHARS
CHARS.SEG	-	CHARS.BIN
:LINK	0.2	CROSSHATCH
CROSSHATCH.SEG	-	CROSSH.SEG
DISPATCH.BIN	1.1	DISPAT.BIN
DTST.BIN	1.2	DTST.BIN
DUAL.BIN	1.0	DUAL.BIN
HIGH.BIN	1.0	HIGH.BIN
JUMP.BIN	1.0	JUMP.BIN
:LINK	1.5	KEYTEST
KEYTEST.SEG	-	KEYTST.SEG
LOOP.BIN	0.0	LOOP.BIN
LOW.BIN	1.0	LOW.BIN
MEM.BIN	2.0	MEM.BIN
NEXTOP.BIN	1.1	NEXTOP.BIN
NXTI.BIN	1.1	NXTI.BIN
PART.BIN	1.1	PART.BIN
PBT.BIN	1.1	PBT.BIN
RAT.BIN	1.3	RAT.BIN
REGT.BIN	1.1	REGT.BIN
SHIFT.BIN	1.1	SHIFT.BIN
STACK.BIN	1.0	STACK.BIN
TST.BIN	1.1	TST.BIN
:LINK	2.1	TESTFLOPPY
TESTFLOPPY.SEG	-	TSTFPY.SEG

>PASCAL.SOURCE - PASCAL COMPILER SOURCES

file name	version	file name on floppy
	short description.	
PASCAL.PAS	6.0	PASCAL.PAS
	Pascal compiler global definitions.	
PAS0.PAS	-	PAS0.PAS
PAS1.PAS	-	PAS1.PAS
PAS2.PAS	-	PAS2.PAS
QCODES.DFS	-	QCODES.DFS
	Q-Code const definitions.	
COMPINIT.PAS	-	COMPIN.PAS
	Initialization.	
CODEGEN.PAS	-	CODEGE.PAS
	Code generator.	
DECPART.PAS	-	DECPAR.PAS
	Declaration processor.	
DEC0.PAS	-	DEC0.PAS
DEC1.PAS	-	DEC1.PAS
DEC2.PAS	-	DEC2.PAS
BODYPART.PAS	-	BODYPA.PAS
	Procedure/function/program body processor.	
BODY0.PAS	-	BODY0.PAS
BODY1.PAS	-	BODY1.PAS
BODY2.PAS	-	BODY2.PAS
BODY3.PAS	-	BODY3.PAS

>PASCAL.BINARY - PASCAL COMPILER SEG FILES

file name	version	file name on floppy short description.
:LINK	6.0	PASCAL
PASCAL.SEG	-	PASCAL.SEG
COMPINIT.SEG	-	COMPIN.SEG
CODEGEN.SEG	-	CODEGE.SEG
DECPART.SEG	-	DECPAR.SEG
BODYPART.SEG	-	BODYPA.SEG
EXPEXPR.SEG	-	EXPEXP.SEG
PASCAL.SYNTAX	-	PASCAL.SYN
PASCAL.RESWORDS	-	PASCAL.RES
LEX.SEG	0.0	LEX.SEG
FQCODES.SEG	1.0	FQCODE.SEG
FRESWORDS.SEG	1.0	FRESWO.SEG
FSYNTAX.SEG	1.0	FSYNTA.SEG
:LINK	2.0	QDIS
QDIS.SEG	-	QDIS.SEG
QCODES	-	QCODES

>PASCAL.MORE.SOURCE - MORE PASCAL COMPILER SOURCES

file name	version	file name on floppy short description.
EXPEXPR.PAS	-	EXPEXP.PAS Expression expansion.
EXPR0.PAS	-	EXPR0.PAS
EXPR1.PAS	-	EXPR1.PAS
EXPR2.PAS	-	EXPR2.PAS
EXPR3.PAS	-	EXPR3.PAS
FSYNTAX.PAS	1.0	FSYNTA.PAS Program to generate PASCAL.SYNTAX from SYNTAX.DAT.
SYNTAX.DAT	-	SYNTAX.DAT Error message data file.
QDIS.PAS	2.0	QDIS.PAS Q-Code disassembler. Also needs QCODES.PAS.
FQCODES.PAS	1.0	FQCODE.PAS Program to generate QCODES from QCODES.DAT.
QCODES.DAT	-	QCODES.DAT Q-Code name data file.
FRESWORDS.PAS	1.0	FRESWO.PAS Program to generate QCODES from QCODES.DAT.
RESWORDS.DAT	-	RESWOR.DAT Q-Code name data file.
LEX.PAS	0.0	LEX.PAS Lexical scanner for compiler.

>DEMO.SOURCE - DEMONSTRATION PROGRAM SOURCES

file name	version file name on floppy short description.
KAL.PAS	- KAL.PAS Kaleidoscope display.
KINETIC.PAS	- KINETI.PAS Demonstrate random raster-ops.
LIFE.PAS	- LIFE.PAS The game of life.
LINE.PAS	- LINE.PAS Line drawing display.
PETAL.PAS	- PETAL.PAS Cycloid drawing display.
MULDIV.PAS	- MULDIV.PAS Double precision multiply and divide for Petal.
SLEEP.PAS	- SLEEP.PAS Sleep for a specified period of time.
SCREENDUMP.PAS	0.0 SCRDMP.PAS Print an image of the screen to an HP 7310A printer.
SEISMO.PAS	- SEISMO.PAS Multi-pen chart recorder display.
SKETCH.PAS	- SKETCH.PAS Sketch on the screen using the tablet.

>DEMO.BINARY - DEMONSTRATION PROGRAM SEG FILES

file name	version	file name on floppy short description.
:LINK	-	KAL
KAL.SEG	-	KAL.SEG
:LINK	-	KINETIC
KINETIC.SEG	-	KINETI.SEG
:LINK	-	LIFE
LIFE.SEG	-	LIFE.SEG
:LINK	-	LINE
LINE.SEG	-	LINE.SEG
:LINK	-	PETAL
PETAL.SEG	-	PETAL.SEG
MULDIV.SEG	-	MULDIV.SEG
SLEEP.SEG	-	SLEEP.SEG
:LINK	0.0	SCREENDUMP
SCREENDUMP.SEG	-	SCRDMP.SEG
:LINK	-	SEISMO
SEISMO.SEG	-	SEISMO.SEG
:LINK	-	SKETCH
SKETCH.SEG	-	SKETCH.SEG

>DEMO.SIGGRAPH.SOURCE - SIGGRAPH 80 DEMONSTRATION SOURCES

file name	version file name on floppy short description.
INITDEMO.PAS	- INITD.PAS Initialize the demo.
PETALDEMO.PAS	- PETALD.PAS Cycloid drawing display.
LINEDEMO.PAS	- LINED.PAS Line drawing display.
LIFEDEMO.PAS	- LIFED.PAS The game of life.
SEISDEMO.PAS	- SEISD.PAS Multi-pen chart recorder display.
GETSAVE.PAS	- GETSAV.PAS Get display from file to screen or save from screen to file.
CREATEWIN.PAS	- CREWIN.PAS Create entry in Screen package's window table.
SLIDER.PAS	- SLIDER.PAS Slide a window from one position to another on the screen.
JUST.PAS	- JUST.PAS Justify text with various fonts in a window.
WIPEWIN.PAS	- WIPWIN.PAS Wipe a picture into a window.
SNOOZE.PAS	- SNOOZE.PAS Pause for a specified period of time.
	Modules for SigGraph demo: MulDiv, Sleep, WindowLib, SigUtils, FontStuff.
WINDOWLIB.PAS	- WINLIB.PAS Window routine library.
SIGUTILS.PAS	- SIGUTI.PAS SigGraph 80 demo utilities.
FONTSTUFF.PAS	- FONTST.PAS Load and unload fonts for Just.

>DEMO.SIGGRAPH.BINARY - SIGGRAPH 80 DEMONSTRATION BINARY FILES

file name	version	file name on floppy short description.
:LINK	-	INITDEMO
INITDEMO.SEG	-	INITD.SEG
:LINK	-	PETALDEMO
PETALDEMO.SEG	-	PETALD.SEG
:LINK	-	LINEDEMO
LINEDEMO.SEG	-	LINED.SEG
:LINK	-	LIFEDEMO
LIFEDEMO.SEG	-	LIFED.SEG
:LINK	-	SEISDEMO
SEISDEMO.SEG	-	SEISD.SEG
:LINK	-	GETSAVE
GETSAVE.SEG	-	GETSAV.SEG
:LINK	-	CREATEWIN
CREATEWIN.SEG	-	CREWIN.SEG
:LINK	-	SLIDER
SLIDER.SEG	-	SLIDER.SEG
:LINK	-	JUST
JUST.SEG	-	JUST.SEG
:LINK	-	WIPEWIN
WIPEWIN.SEG	-	WIPWIN.SEG
:LINK	-	SNOOZE
SNOOZE.SEG	-	SNOOZE.SEG
WINDOWLIB.SEG	-	WINLIB.SEG
SIGUTILS.SEG	-	SIGUTI.SEG
FONTSTUFF.SEG	-	FONTST.SEG
FEATURES.SLIDE	-	FEATUR.SLI
SOFTWARE.SLIDE	-	SOFTWA.SLI
NETWORK.SLIDE	-	NETWOR.SLI
UCODE.SLIDE	-	UCODE.SLI
IO.SLIDE	-	IO.SLI
3RCC.SLIDE	-	3RCC.SLI
JUST.DEMO	-	JUST.DEM
GRAPH.PIC	-	GRAPH.PIC
3RCC.PIC	-	3RCC.PIC
BLANK.PIC	-	BLANK.PIC
WASHDC.PIC	-	WASHDC.PIC
NGR13.KST	-	NGR13.KST
MET22.KST	-	MET22.KST
DEMO	-	DEMO
DEMO1.CMD	-	DEMO1.CMD

>PERQFILE.USERLIBRARY - USER LIBRARY SOURCES

file name	version	file name on floppy short description.
CURSDESIGN.PAS	1.0	CURSDE.PAS Program used to design new cursors.
FONTED.PAS	-	FONTED.PAS Program used to create new fonts.
FONT2ED.PAS	-	FONT2E.PAS Part of Fonted.
FONTED.CURSOR	-	FONTED.CUR File of pictures needed by the font editor.
DR.MEMORY.PAS	-	DRMEMO.PAS Peek and poke into the memory manager tables.
TD.PAS	-	TD.PAS Demo of Grapics.
TD1.PAS	-	TD1.PAS Part of TD.
TD2.PAS	-	TD2.PAS Another part of TD.
MAZE.PAS	-	MAZE.PAS Draws a maze and runs a mouse through it.
GENMAZE.PAS	-	GENMAZ.PAS Creates a well-formed random maze.
MAZEPLAYER.PAS	-	MAZEPL.PAS Used by Maze to allow user to try to get through the maze.

PERQ Fault Dictionary

The Key to the PERQ Diagnostic Display

Three Rivers Computer Corporation
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250

Display	Description
000	Boot never got going, StackReset doesn't work or other major problem in the processor board (or clock).
001	Simple Branches fail.
002	Main Data Path Failure.
003	Dual Address failure on Registers.
004	Y Ram Failure.
005	Const/Carry Propogate failure.
006	ALU failure.
007	Conditional Branch failure.
008	Looping failure.
009	Control Store (or Write Control Store) failure.
010	Hung in Disk Boot.
011	Memory Data Error.
012	Memory Address Error.
013	Disk never became ready.
014	Couldn't boot from either disks.
015 - 020	Bad Interrupts Reading Floppy Disk Data.
030	VFY Hung.
050	Bad Error Message from VFY.
051	Empty stack bit not working.
052	Could not load TOS.
053	Push did not work.
054	Stack Empty did not go off.
055	Data error in push.
056	Empty or Full set when that is not the case.
057	Data error in bit 15 of the stack.
058	Stack empty set when the stack is full.
059	Data error on stack.
060	Data error after POP. Bit 14.
061	Data error after POP. Bit 13.
062	Data error after POP. Bit 12.
063	Data error after POP. Bit 11.
064	Data error after POP. Bit 10.
065	Data error after POP. Bit 9.
066	Data error after POP. Bit 8.
067	Data error after POP. Bit 7.
068	Data error after POP. Bit 6.
069	Data error after POP. Bit 5.
070	Data error after POP. Bit 4.
071	Data error after POP. Bit 3.
072	Data error after POP. Bit 2.
073	Empty wrong.
074	Data error after POP. Bit 1.
075	Data error after POP. Bit 0.
076	Empty not set after all pops.
077	Call test falied.
078	Odd didn't jump on a 1.
079	Odd jumped on a 0.

Display	Description
080	Byte sign didn't jump on 200.
081	Byte sign jumped on 0.
082	C19 didn't jump when it should have.
083	BCP[3] didn't jump when it should have.
084	C19 jumped when it shouldn't have.
085	BCP[3] jumped when it shouldn't have.
086	GTR didn't jump.
087	GTR jumped when it shouldn't have.
088	GEQ didn't jump.
089	GEQ jumped when it shouldn't have.
090	LSS didn't jump when it should have.
091	LSS jumped when it shouldn't have.
092	LEQ didn't jump.
093	LEQ jumped when it shouldn't have.
094	GEQ didn't jump on equal.
095	LEQ didn't jump on equal.
096	Carry didn't jump when it should have.
097	Carry jumped when it shouldn't have.
098	Overflow didn't jump when it should have.
099	Overflow jumped when it shouldn't have.
100	And-Not ALU function failed.
101	Or ALU function failed.
102	Or-Not ALU function failed.
103	And ALU function failed.
104	Or-Not ALU function failed.
105	Not-A ALU function failed.
106	Not-B ALU function failed.
107	Xor ALU function failed.
108	Xnor ALU function failed.
109	OldCarry-Add ALU function failed.
110	OldCarry-Sub ALU function failed.
111	OldCarry-Add /w No OldCarry failed.
112	Fetch error on Force Bad Parity.
113	Unexpected Parity error.
114	No parity errors on force bad parity.
115	Wrong address on force bad parity.
116	Upper 4 bit test failed.
117	MDX test failed.
118	Stack upper bits test failed.
119	Store/Fetch test failed.
120	Unexpected refill.
121	BPC test failed.
122	Fetch4 test failed.
123	Fetch4R test failed.
124	Store4 test failed.
125	Fetch2 test failed.
126	Store2 test failed.
127	NextOp test failed.
128	Fetch/Store overlap failed.

Display	Description
129	Bad interrupt Loc 4.
130	Bad interrupt Loc 14.
131	Bad interrupt Loc 20.
132	Bad interrupt Loc 30.
133	Data error on memory sweep.
134	Address error on memory sweep.
135	Field didn't work.
136	Dispatch did not jump.
137	Wrong Dispatch target.
138	Data error on inverted memory sweep.
139	Address error on inverted memory sweep.
150	Sysb not loaded correctly or hung.
151	Sysb did not complete.
152	Illegal Boot Key.
153	Hard Disk Restore Failure.
154	No such boot.
155	No interpreter for that key.
156	Interpreter file is empty.
157	Disk Error.
158	Floppy error.
159	Malformed Boot File.
160	CheckSum error in microcode.
161	CheckSum error in QCode.
162 - 168	Bad interrupts.
198	QCode interpreter microcode not entered correctly.
199	System not entered - calls or assignments don't work.
200	System entered, InitMemory to be called.
201	InitMemory entered.
203	SAT and SIT pointers set.
204	StackSegment number set.
205	Reading the BootBlock.
206	System version number set.
207	Head of free-segment-number list set.
208	First system segment number set.
209	System boot disk set.
210	System boot character set.
211	Boot block read.
212	Default heap segment number set.
213	First used segment number set.
214	Before setting freelists of data segments.
215	Before trying to allocate a segment number.
216	Temporary segment number allocated.
217	Ready to enter loop to find memory size.
218	Exited from memory size loop.
219	Restored mangled word.
220	Released temporary segment number.

Display	Description
221	Located segment adjacent to the I/O segment.
222	Modified the location of I/O segment.
223	Adjusted free memory.
224	Freelists of data segments set.
225	Set screen segment.
226	Header buffer allocated for swapping.
227	Status buffer allocated for swapping.
228	SwappingAllowed set false.
229	All boot-loaded segments set UnSwappable (if booted from floppy), InitMemory complete, ready to return to System.
230	Starting to increase number of segments allowed (because memory is larger than 1/4 megabyte).
231	Changed maximum of SITSeg.
232	Changed size of SITSeg.
233	Changed maximum of SATSeg.
234	Changed size of SATSeg.
235	Created new unallocated segment numbers.
236	Finished InitMemory.
300	InitIO to be called.
301	InitIO entered.
302	KeyEnable set false.
303	Buffers allocated.
310	InitInterruptVectors to be called.
320	InitInterruptVectors complete, InitDeviceTable to be called.
322	Starting to initialize ETHERNET.
325	ETHERNET initialized.
330	InitDeviceTable complete, InitScreen to be called.
340	InitScreen complete, InitTablet to be called.
350	InitTablet complete, InitCursor to be called.
360	InitCursor complete.
361	Interrupts are now off; about to send device table to microcode.
363	Got control block for Z80 speech.
364	Set up video registers.
365	Screen is now started.
366	Got control blocks for keyboard, RS232, and GPIB.
368	Microcode returned.
369	Interrupts are now turned on.
370	Microcode informed that the device table has been initialized, IO microcode initialization complete, LocateDskHeads to be called.
371	LocateDskHeads entered, buffers allocated.
373	Disk heads at cylinder 0 or disk broken.

Display	Description
374	Disk heads at cylinder 0 (not broken).
375	Microcode instructed to consider current position as cylinder 0.
376	Dummy read of cylinder 0, sector 0 complete, about to dispose buffers and exit LocateDskHeads.
380	LocateDskHeads complete, FindSize to be called.
381	FindSize entered and buffers allocated.
382	Disk access attempt returned.
383	Size of disk determined, about to dispose buffers and exit FindSize.
390	FindSize complete.
400	Keyboard enabled.
410	InitGPIB to be called.
411	InitGPIB entered, buffers allocated.
412	First GPIB command built.
413	First GPIB command sent to Z80.
414	Second GPIB command built.
415	Second GPIB command sent to Z80, about to dispose buffers and exit InitGPIB.
420	InitGPIB complete.
499	Clock enabled, about to exit InitIO.
500	InitIO complete, InitStream to be called.
600	InitStream complete, FSInit to be called.
700	FSInit complete.
800	Command file and Console opened, InitExceptions to be called.
810	InitExceptions complete.
820	System version number set.
822	Current 60 Hz. clock value read.
824	60 Hz time reference set, TimeStamp time reference to be set.
900	FSSetUpSystem to be called.
950	FSSetUpSystem complete.
951	About to enable swapping (if booted from hard disk).
952	FSLocalLookup and EnableSwapping complete.
999	System fully initialized, system title line to be printed.

Three Rivers Computer

SOFTWARE REPORT FORM

SOF-1 Orig. 5-81

Reference # (Supplied by Submitter)

For 3RCC Use Only:	REQ.#:	Date Recd:	Date Processed:

FROM

Person to Contact

Company Name

Customer Address

City/State/Country ZIP

Phone Date

HARDWARE

PERQ Model # _____

PERQ Serial # _____

Memory Size _____

Options:

Floppy ☐ Network ☐ WCS ☐

DISK SIZE

☐ 12

☐ 24

☐ 48

O.S. Version # _____

Subsystem, Program, Module or Manual Name: _____

Version # _____

REPORT TYPE

- ☐ Logic/Coding Error
- ☐ Documentation Error (Page #_____)
- ☐ Suggestion for Improved Efficiency
- ☐ Request for Software Modification
- ☐ Inquiry
- ☐ For Your Information

PRIORITY

- ☐ CRITICAL
- ☐ HIGH
- ☐ STANDARD
- ☐ LOW

Program demonstrating problem attached:

☐ on floppy (preferred) ☐ hard copy ☐ Not Avail. Not Applic

Can this problem be reproduced at will? ☐ Yes ☐ No

DESCRIPTION OF PROBLEM:

CUSTOMER _____ PERQ SERIAL NUMBER _____

PERQ Customer's Report & Reader's Comments

Packing

What was the condition of the packing when you received your PERQ? Please be specific?

_____

_____

_____

_____

Missing Materials

What (if anything) was listed on the packing slip but not received?

_____

_____

_____

PERQ's Overall Condition

Please describe any damage(s) to the PERQ system when received. Be specific.

BASE _____

_____

_____

DISPLAY _____

_____

_____

KEYBOARD _____

_____

_____

TABLET (BIT PAD) _____

_____

_____

OTHER _____

_____

_____

Clarity of Installation Instructions

Were you able to easily unpack and install the system?

_____

_____

_____

_____

Please note below any suggestions or comments which you feel may be beneficial to us in better serving our customers.

Please designate below a software and hardware person in your organization whom we can contact for information or problems.

Name _____ Title _____

SOFTWARE CONTACT

Company/Organization _____ Telephone Number _____

Address _____

Address _____

Name _____ Title _____

HARDWARE CONTACT

Company/Organization _____ Telephone Number _____

Address _____

Address _____

Please comment on the documentation manual that you received with your PERQ. Did you find it understandable, usable, and well organized? Did you find any errors? If so, identify the page number and text.

Prepared/Completed by: _____ / /
Name Title Date

THANK YOU - THREE RIVERS COMPUTER CORPORATION
720 Gross Street
Pittsburgh, PA 15224
(412) 621-6250