# PERSONAL PASCAL

## For ■ The ■ Atari ■ ST

OSS™ Optimized Systems Software
*A Division of ICD, Incorporated*
1220 Rock Street ■ Rockford, Illinois 61101 (815) 968-2228

# *A Reference Manual For*

# Personal Pascal

## One Pass
## Native Code
## Pascal Compiler

Atari 520/1040 ST Version 2
Developed by J. Lohse



Manual revision by
Bill Wilkinson
and
Earl Rice

# DISCLAIMER

***Neither OSS nor CCD make representations or warranties with respect to the contents hereof and specifically disclaim all warranties of merchantability or fitness for any particular purpose. This manual and the software it describes are subject to change without notice.***

# A CAUTION ! ! !

**If this is the first time** you've ever used Pascal, *PLEASE* don't try to learn the language from this manual. It's not designed as an introductory Pascal tutorial, and will only make you frustrated (and probably mad at us). **If you are a beginner,** we suggest that you look in a local bookstore for the following book:

**Oh! Pascal!**
*Doug Cooper and Michael Clancy*
W W Norton and Company.

This is the best tutorial we have found. Further, the terms used are generally the same as in the Language Reference section of this manual.

**If you are an experienced programmer** looking for algorithms, we recommend:

**Software Tools in Pascal**
*Kernigan and Plauger*
Addison-Wesley

**Algorithms**
*Robert Sedgewick*
**Addison-Wesley**

# ABOUT TECHNICAL SUPPORT

Technical support is a costly part of our business. We are determined to give the best support we can to programmers who buy Personal Pascal. Our ability to do that rests upon two suppositions. The first is that you already know how to program in Pascal and that you are genuinely stumped even after referring to your manual. The second supposition is that you have actually bought Personal Pascal legitimately and are entitled to our full support.

Given these suppositions we require two things of you. First, if you must contact OSS with a technical question, we strongly recommend that you call our bulletin board system, since you can leave an example program along with your question and thus save both you and OSS time and money. Second, we will ask you to give us your registration number ( with your message on the BBS) before receiving any technical assistance.

That means you *must* mail in your registration card, because we will only give you technical support if your number is on file with us.

If you do not have a modem, or if the program in question is very large, please send a disk and a description of the problem to:

**OSS /ICD, Inc.**
**1220 Rock Street**
**Rockford, Illinois 61101**

(815) 968-2228 BBS: (815) 968-2229 FAX: (815) 968-6888

# PREFACE

This is the second edition of the Personal Pascal Reference Manual. In addition to presenting the new features of Personal Pascal version 2, it incorporates a new layout and an index. Quite frankly, we've been unhappy with the old Personal Pascal manual from the beginning. Everything was there, but the lack of an index and the general arrangement of the manual begged for improvement. We also got complaints about the "Backus Naur Formalism" (BNF) we used to present syntax.

When we started to update the manual, we were ready to radically re-design it. Good sense prevailed, however, and we were less rash than we thought we might be. The sequence of the chapters is more in the order that Pascal does things. We've simplified explanations and added examples. The longer explanations are in the appendix, command synopses are more task-oriented, and the BNF is replaced by syntax diagrams. We hope this will make life easier for you as you program in Personal Pascal.

While this manual is not intended to teach anyone Pascal, we have recognized the needs of the occasional programmer and included programming examples in the appendix. This should also help the experienced programmer become familiar with the general use of GEM from Personal Pascal.

We hope you enjoy this manual.

# WHERE TO LOOK

The Personal Pascal Reference Manual is divided into 9 sections:

**SECTION 1, GETTING STARTED**, describes the hardware required to write programs in Personal Pascal, and tells you how to get up and running.

**SECTION 2, THE MANAGER**, introduces you to the Personal Pascal Manager. The Manager coordinates Personal Pascal's functions to make writing and compiling programs easier. This section shows you what the Manager does, and how to use it to configure your programming environment to suit your own system and preferences.

**SECTION 3, THE EDITOR**, describes the powerful GEM-based Personal Pascal Editor. The ability to cut and paste among as many as 3 files at once makes short work of program modification and maintenance.

**SECTION 4, THE COMPILER**, explains the compiler and its options.

**SECTION 5, THE LINKER**, describes the Personal Pascal Linker and its use.

# PREFACE

This is the second edition of the Personal Pascal Reference Manual. In addition to presenting the new features of Personal Pascal version 2, it incorporates a new layout and an index. Quite frankly, we've been unhappy with the old Personal Pascal manual from the beginning. Everything was there, but the lack of an index and the general arrangement of the manual begged for improvement. We also got complaints about the "Backus Naur Formalism" (BNF) we used to present syntax.

When we started to update the manual, we were ready to radically re-design it. Good sense prevailed, however, and we were less rash than we thought we might be. The sequence of the chapters is more in the order that Pascal does things. We've simplified explanations and added examples. The longer explanations are in the appendix, command synopses are more task-oriented, and the BNF is replaced by syntax diagrams. We hope this will make life easier for you as you program in Personal Pascal.

While this manual is not intended to teach anyone Pascal, we have recognized the needs of the occasional programmer and included programming examples in the appendix. This should also help the experienced programmer become familiar with the general use of GEM from Personal Pascal.

We hope you enjoy this manual.

**SECTION 6, PERSONAL PASCAL**, describes Pascal program structure in general, as well as Personal Pascal's Definitions and extensions to the ISO Pascal standard.

**SECTION 7, GEM**, introduces the GEM concepts you need to write professional programs. Alert and Message Boxes, Menus, Windows and Events are all covered specifically and thoroughly.

**SECTION 8, THE APPENDIX**, is a treasure chest of techniques and information to help you develop the full power of the Atari ST computer while avoiding known pitfalls. This section contains tables, examples, and tested program shells to speed your programming.

**SECTION 9, THE INDEX**, cross-references tables, examples, key words and concepts throughout the reference manual.

# TABLE OF CONTENTS

## Table of Contents ( Continued )

## Table of Contents ( Continued )

## Table of Contents ( Continued )

# GETTING STARTED

## REQUIRED HARDWARE

Personal Pascal will work on any Atari ST computer. The minimum system configuration is a 520ST with a single-sided disk drive. Developing programs on a minimum system requires careful planning, but is not difficult as long as your programs aren't too long. The section explaining *The Manager* contains helpful suggestions for writing programs on a single-drive system and for using a RAM disk with Personal Pascal.

If you have a 1 Meg or larger memory in your ST, a RAM disk makes the Personal Pascal system *very* fast. Just remember to copy your work from the *RAM* disk to a *physical* disk before you turn off your computer!

If you have a Meg of RAM and a hard disk drive, then the Personal Pascal system *really* shines! High speed and large disk capacity easily handle even the largest programs. This combination is the ideal program development system; practically Programmer Heaven!

# MAKE A BACK-UP!

**This is your OSS House Mother talking! Listen up!**

Before you do anything else, *even before you finish this page*, **MAKE A BACK-UP COPY OF YOUR PERSONAL PASCAL DISKETTE!** Be sure to open the write-protect tab first! *Did you change your sox this morning?*

Ok. Now that you've made a back-up copy, ( You *DID* make a back-up copy, didn't you??!! ) put it in a box, mark it "*DANGER!! RADIOACTIVE WASTE!!*," and hide it on the top shelf of your closet! Better yet, put it in a lead-lined, magnetically-shielded underground concrete vault!!

Are we paranoid? Yep. You wouldn't *believe* how many panicky phone calls we get from people who have just trashed their master diskettes. Did you know it takes *at least three days* to get a new diskette to you? You send the old one to us, *plus money*. When we get the old one, we send the new one to you. Three days... If everything works right...

You've been warned. If you don't make a back-up now, you're hopelessly masochistic. If you have to call us because you messed up your only copy of Personal Pascal, we'll probably *laugh* at you, and we'll *NEVER* let you live it down! After all, we didn't leave Personal Pascal unprotected so *pirates* could get it! We did it so **YOU CAN MAKE A BACK-UP COPY! SO DO IT!!**

# RUNNING PERSONAL PASCAL

Running Personal Pascal is easy. Put your back-up copy into your disk drive and get a directory. you should see the following files and folders in the directory listings of your Personal Pascal diskettes.

| | |
|---|---|
| **PASCAL .PRG** | { a program file } |
| **EDITOR .PRG** | { a program file } |
| **COMPILER.PRG** | { a program file } |
| **LINKER .PRG** | { a program file } |
| **INFO** | { a folder } |
| **DEMOS** | { a folder } |

The **INFO** folder contains additions, including any bugs discovered since the printing of this manual. You can "Show" or "Print" any file in this folder.

The **DEMOS** folder contains demonstrations of the GEM/Pascal Library and other Personal Pascal programming examples.

**PASCAL.PRG** is the Personal Pascal Manager. Double-click on this file to enter the Personal Pascal system. Don't double-click on the other .PRG files, since they're not designed to be run from the desktop.

Once you are in the Manager environment, you can use Personal Pascal's programming facilities. Turn to the appropriate section for instructions.

*Happy Programming!*

BLANK PAGE

# THE MANAGER

## WHAT THE MANAGER DOES

The Manager coordinates Personal Pascal's editing, compiling and linking functions. It also allows you to configure your programming environment to suit your preferences and your hardware. Once you establish compiler and linker options and tell the manager what source files you are using, you can save your working set-up to disk, and the Manager will maintain your programming environment.

## MANAGER COMMANDS

The Manager's commands can be invoked using either the mouse or the keyboard. All Menu functions are duplicated by keyboard commands using a single letter or a function key.

**SEE:** *Manager Command summary* page 2-15.

The Manager's dialog boxes require the use of the mouse to effect most changes, but allow the use of the keyboard where possible.

## CHECK-MARKS

When you invoke a menu option, the Manager puts a check-mark next to it in the menu. The check-mark provides an easy way to remember what you did last during a programming session.

# THE FILE MENU

The File menu allows you to run the *Editor, Compiler, Linker,* or a compiled program. Select the function you want from the menu, then click on the appropriate file when the *Item Selector* dialog box appears.

**EDIT** runs the Editor.

**COMPILE** runs the Compiler.

**COMPILE ALL** compiles all the source files you have selected to be compiled in the current session.
**SEE:** *Set Source Files.*

**LINKER** runs the Linker.

**RUN PROGRAM** presents the Item Selector to choose a program to run.

**QUIT** returns you to the desktop.

**NOTE:** The default file is the last program edited, compiled, linked or run.

# THE OPTIONS MENU

The *Options* menu lets you set compiler and linker options. It also allows you to define a source file list for the compiler, and locate Personal Pascal's programs to suit your hardware.

**COMPILER OPTIONS**
**SEE:** Page 2-4

**LINK OPTIONS**
**SEE:** Page 2-8

**SET SOURCE FILES**
**SEE:** Page 2-9

**LOCATE PROGRAMS**
**SEE:** Page 2-10

## LOAD OPTIONS

All the options you select, including program locations, can be saved to any file. The file called called *PASCAL.INF* is loaded when Personal Pascal boots, and configures your system according to your last saved specification. The *Load Options* function lets you load your default configuration, or any other configuration at any time. This lets you configure your system for several projects and choose among them easily.

## SAVE OPTIONS

When you have your system set to your preference, click on *Save Options* to save its current configuration to any file. If you save to *PASCAL.INF*, your system will boot up configured as you defined it.

# COMPILER OPTIONS

The Compiler is flexible enough to take care of many of the details involved in putting together different types of programs. Selectable options reduce the number of Compiler directives you need to remember and use.

**PROGRAM TYPE**
You can compile a GEM program, an ACCessory, or a TOS program having no GEM functions . Click the applicable radio button for the type of program you are compiling. If your program is not an accessory and uses any of the GEM/Pascal library functions, you should leave the GEM option selected. Accessories assume the use of the GEM/Pascal library

**NOTE:** There are a few special considerations when writing desk accessories.
**SEE:** *Writing Desk Accessories*, in the appendix.

**PAUSE AFTER ERRORS**
When the compiler encounters an error, it puts an identifying error number and message into a file, using the name of the file being compiled, but appending the extension *.ERR*. With *Pause After Errors* selected,

an alert box will appear on the screen, asking whether you want to continue the compilation, cancel it and return to the Manager, or cancel and go directly to the Editor. If you choose to cancel and go to the editor, you will see the cursor marking the the error's position, and an error message. When you correct the error, you can run the Compiler directly from the Editor.

### CHAIN TO LINKER

When you check *Chain to Linker*, the compiler will automatically run the linker when your program is successfully compiled. The linker will use the program type you have specified, GEM, TOS, or ACC, to link the appropriate Personal Pascal files to your program.

## FULL DEBUG MODE

When the *Full Debug Mode* box is checked, the compiler will include code to support debugging. If there is an error while your program is running, its execution will stop and you will be given information about the error, including:

**An error description**
**The subprogram name**
**The line containing the error**
**The current Program Counter value.**

**NOTE:** If an error occurs in the operating system, the subprogram name *UNKNOWN* will be displayed.

## CLEAR VARIABLES

If the *Clear Variables* box is checked, the compiler will generate code to clear all local variables, function return values, and space returned by the New procedure.

## STACK AND POINTER CHECKING

When the *Stack and Pointer Checking* option is checked, the compiler will generate code to check the

stack and pointers against overflow at run-time. If such an error should occur while your program is running, its execution will stop and an alert box or a text line ( for TOS programs ) will appear, describing the error. If you are in full debug mode as well, you will be given additional information about the error.

**SEE:** *Full Debug Mode*, page 2-6.

**RANGE CHECKING**

When the *Range checking* option is checked, the compiler will generate code to check subranges and array indices at run-time. If a range error occurs while your progam is running, execution will stop and an alert box or a text line ( for TOS programs ) will appear, describing the error. If you are in full debug mode as well, you will be given additional information about the error.

**SEE:** *Full Debug Mode*, page 2-6.

**PRINT LISTING**

When this option is checked, the compiler prints a listing to the screen as it compiles your program. This option defaults to its *UNCHECKED* state.

# LINKER OPTIONS

**PROGRAM TYPE**
As with the compiler, you can tell the Linker to link the appropriate GEM/Pascal files for a GEM program, a TOS program, having no GEM functions, or an ACCessory. Click the applicable radio button for the type of program you are linking. If your program uses any of the GEM/Pascal library functions, you should leave the GEM option selected. Accessories assume the use of the GEM/Pascal library.

**NOTE:** There are a few special considerations when writing desk accessories.
**SEE:** *Writing Desk Accessories*, in the appendix.

**ADDITIONAL LINK FILES**
The linker automatically links Pascal object files and Pascal library files to produce your program. If you used the Modular compiler directive, {$M}, you can enter the names of additional files that you want linked, such as assembly language modules, or other Pascal object files. You can enter as many file names, separated by commas, as will fit on the lines given.

**NOTE:** There is one restriction; a filename may not extend from the first line onto the second line.

**LINK COMPILED OBJECT FILE**
When this box is checked, the Linker prompts you for the name of the file you want linked. The default filename is the last file edited, compiled or linked in the current session. As an example; if you had just compiled a module and wanted to link your master object file first, you would use this option.

# MANAGER OPTIONS

## SET SOURCE FILES

This menu option lets you define a list of programs to compile. Its dialog box presents two lists. The left list is a directory of available files. You can change disk drives by clicking the appropriate radio buttons; you can change the path by editing the path name at the top of the dialog box; you can select a file by clicking the filenames, using the close box to go up a level in the directory path; and you can add a highlighted filename to the compile list by clicking the *Transfer* button.

**NOTE:** The *.** button toggles the path between *.PAS* and **.**, to show files with any extender.

The right list shows the files you want to compile. To remove a filename from the list, click on it to highlight it, then click on the *Erase* button. When your list is complete, click the *OK* button. The *Cancel* button closes the dialog without saving the compile list to memory.

## LOCATE PROGRAMS

This option lets you specify the locations of the Personal Pascal program and library files. You can configure your system to expect certain files to be on disk A, others on disk B, etc. You can even change the names of the programs that Personal Pascal runs as the Editor, Compiler or Linker, though we don't usually recommend it.

*Locate Programs* presents a dialog box containing a directory window, drive selection buttons, and text lines for path and file names. To locate a file, select the file to be located, using the mouse in the the directory window, or by typing the path and file names on the appropriate lines. Click the *OK* button to save the file location, or the *Exit* button to leave the dialog without locating the file.

**NOTE:** For all files except *PASLIB* and *PASGEM*, the .* button toggles the path between *.* and *.PAS* to show files with any extender.

# SPECIAL FUNCTIONS MENU

The Special menu gives you access to the file functions, *Copy, Rename, Erase* and *Print*, without having to exit to the desktop. These options present a dialog box containing a directory window, drive selection buttons, and text lines for path and file names.

## COPY

The *Copy* option allows you to copy a file from one directory or disk to another, much as from the desktop.

The *Copy Files* dialog box presents two directory lists, one for a source directory, and one for a destination directory. You can change disk drives by clicking the appropriate radio buttons; you can change file paths by editing the path names at the top of the dialog box, you can select a file by clicking the filenames and close box, and you can copy a highlighted file from the *Source* directory to the *Destination* directory by clicking the *Copy* button.

Clicking the *Exit* button leaves the dialog without copying a file.

**RENAME**
To rename a file, select the file to be renamed, using the mouse in the directory window, or by typing the path and file names on the appropriate lines in the dialog box. Click the *Rename* button to rename the file, or the Exit button to leave the dialog without renaming the file.

**ERASE**
To erase a file, select the file to be erased, using the mouse in the directory window, or by typing the path and file names on the appropriate lines in the dialog box. Click the *Erase* button to erase the file, or the *Exit* button to leave the dialog without erasing the file.

**PRINT**
To print a file, select the file to be printed by using the mouse in the directory window, or by typing the path and file names on the appropriate lines in the dialog box. Click the *Print* button to print the file, or the *Exit* button to leave the dialog without printing the file.

# CONFIGURING YOUR SYSTEM

Thanks to the *LOCATE FILES* option of the Personal Pascal manager, it is easy to use this package on any Atari ST system.

Personal Pascal version 2 is large enough that you can not have all the required files (Manager, Editor, Compiler, Linker, include files, libraries) on a single diskette, so it is supplied on a pair of single sided 3.5" diskettes, compatible with either single sided or double sided disk drives.

The implication of this is that owners of single drive systems will have to swap diskettes during the compile process. We have attempted to set up the files on these disks in a way optimized for owners of single drive 520ST systems. In particular, we have placed the major program modules (Editor, Compiler, Linker) on the "B" disk. The source modules (include files) are on the "A" disk. Other files have been placed on the disks where space is available, in a logical fashion.

We recommend that single drive users make a fresh copy of the "A" disk for each Pascal project that they undertake. The "B" disk may be write protected in most circumstances. TOS will prompt you to swap these diskettes at the appropriate times.

If you have two single sided drives, we recommend that you put the two diskettes ("A" and "B") into the corresponding drive. Again, you should use a fresh copy of "A" for each project.

If you have a double sided drive, you may copy all the files from the two diskettes onto one double sided diskette. Make a fresh copy of this diskette for each project.

If you have a RamDisk program, we recommend that you copy as many files as possible from the "B" disk to the RamDisk. If your RamDisk is large enough (e.g., if you have a 1040) you can copy ALL the files to it. However, we do NOT recommend keeping your source on the RamDisk. If the Personal Pascal files are lost from the RamDisk because of power failure or system crash, they can easily be reloaded. If your source is on the RamDisk, it might be lost permanently.

Finally, if you have a hard disk and you are working on several projects, you may wish to put a copy of the manager in each project's subdirectory. Then you can use "Locate Files" to specify that all the required files are (perhaps) in a common directory.

In any case, once you have used "Locate Files" (and other options), you can save your choices in the current directory, and the manager will automatically reconfigure the system each time you enter it.

# COMMAND SUMMARY

Many of the Manager's functions can be invoked both by menu selections and keyboard commands. The following pages present a summary of menu options and their keystroke equivalents.

**FILE MENU**
**SEE:** Page 2-16

**OPTIONS MENU**
**SEE:** Page 2-17

**SPECIAL MENU**
**SEE:** Page 2-18

# FILE MENU

**Edit** **[E]**
Presents a dialog box asking for a source file to edit, then runs the Editor.

**Compile** **[C]**
Presents a dialog box asking for a source file to compile, then runs the Compiler.

**Compile All** **[A]**
Compiles the source files contained in the compile list set up with the *Set Source Files* option of the *Options* menu.
**SEE:** *Set Source Files.*

**Link** **[L]**
Depending upon the Linker options set, presents a dialog box asking for an object file to link to the Personal Pascal library files, then runs the Linker.
**SEE:** *Link Options.*

**Run Program** **[R]**
Presents a dialog box asking for a *.PRG* or *.TOS* file to run.

**Quit** **[Undo]**
Returns to the desktop.

**NOTE:** Pressing the [Alternate] key with E, C, L or R, is equivalent to *Edit, Compile, Link* or *Run Program* using the last invoked filename.

# OPTIONS MENU

**Compiler Options**[F1]
Presents a dialog box displaying available compiler options. Click the appropriate button to select an option. You can also enter a new temporary directory path.
**SEE:** *Compiler Options*, page 4-2

**Linker Options** [F2]

**Set Source Files** [F3]
Presents a dialog box displaying available Linker options. Click the appropriate button to select an option. You can also enter additional link file names.

**Locate Programs** [F4]
Defines the paths used to find Personal Pascal's Editor, Linker, Compiler and library files. The paths can be saved to the *PASCAL.INF* file, using *Save Options*, to make the configuration the new default at boot time, or any other *.INF* filename to make it available to *Load Options.*

**Load Options**[F5]
Loads the Compiler, Linker and program path options you have saved to any *.INF* file.

**Save Options** [F6]
Saves the current Compiler, Linker, program path options, etc., under any *.INF* filename.

# SPECIAL MENU

**Copy [Shift] [F1]**
Presents a dialog box with directory windows to copy a file from one directory or disk, to another.

**Rename [Shift] [F2]**
Presents a dialog box with a directory window to allow renaming of files

**Erase [Shift] [F3]**
Presents a dialog box with a directory window to allow single files to be erased.

**Print [Shift] [F4]**
Presents a dialog box to select a file to be sent to the print device.

# THE EDITOR

You can use any editor or word processor to write your Personal Pascal source code. The only requirement is that it be able to produce an ASCII text file. The Personal Pascal Editor is specifically designed for the programmer. It's a straight-forward editor with easily used features that make writing source code pleasant and uncomplicated. The Editor can load 3 files simultaneously, and its Cut and Paste features make it easy to move blocks of code from one file to another. The Search and Replace functions work quickly, and you can have up to 4 place markers. When you are ready to compile and link, the Editor will invoke those functions for you. Most commonly-used options are selectable by both the mouse and single-character keyboard commands. Overall, the Personal Pascal Editor provides an amiable environment in which to edit text files.

# USING THE EDITOR

To enter the Editor from the Personal Pascal Manager, click on the *Edit* option of the *File* menu. To enter it from the desktop, click *EDIT.PRG.*

The editor will open with an item selection dialog box. If necessary, you can change directories or drives by entering the appropriate specification and clicking the *Close Box* or the *Drag Bar.* Click on the text file you want to edit, or type a file name to create a new file. Click the *OK* button or press [Return] to load or create the selected file.

When the file is loaded, it will be visible in the work window. In the upper right corner of the window is a series of indicators. The left one will display a highlighted numeral 1, indicating that you are currently viewing file space number 1. To its right will be two *Close Box* symbols, showing that file spaces 2 and 3 are empty.

You can load as many as 3 files at the same time. The Editor will put each file into the next available space. The indicators will show you which file space you are viewing, and the name of the file you are viewing will appear in the title bar at the top of the work window. To switch from one file space to another, click on its indicator.

**NOTE:** You can't view an empty file space. To load another file, see: *Open* or *New*, page 3-12.

## ENTERING TEXT

To enter text, simply begin typing. The work window can show a maximum of 76 characters per line. At the end of that line, the work window begins scrolling as you enter more text. The maximum length of a line is 159 characters. When you reach that limit, the Editor stops accepting characters and you must press [Return] or [Backspace]. After you press [Return], you can continue entering text on a new line.

## ENTRY MODE

The Personal Pascal Editor can enter text in either *Insert Mode* or *Overwrite Mode.*

**Insert Mode** causes any text to the right of the cursor to be moved to the right as you type.

**Overwrite Mode** causes text to the right of the cursor to be replaced by the text you enter.

## AUTO-INDENTING

The *Auto Indent* feature of the *Options* menu allows you to properly indent your Personal Pascal programs with a minimum of effort. When this option is selected, the Editor remembers your last tabbed indentation and places the cursor in that column when you press the [Return] key. Each new tab sets a new margin. To back up, use [Backspace] after [Return]. The Editor will use the previous tabbed margin for subsequent text.

## TABS

Set tab intervals using *Tabsize*. Click the option, then enter the interval you want in the dialog box. Click the *OK* button or press [Return] to set the interval to the number shown. You can choose any reasonable tab size, but only a tab size of 8 characters shows correctly when displayed from the GEM desktop.

When *Insert Mode* is selected, Tab causes the cursor to move to the next tab position, pushing any text to its right along with it. In *Replace Mode*, the cursor moves to the Tab position without affecting the text.

## SAVING A FILE

When you are through editing a file, save it under its original name using the *Save* option from the *File* menu, or save it under a new name, using *Save as*. *Save* sends a file to disk storage without further prompting. *Save as* displays the *item selector* dialog box to allow you to select or enter a file name.

## SAVING A TEXT BLOCK

You can save a previously marked block of text to a file by selecting the *Save Block* option from the *Block* menu. The item selector dialog box will appear to allow you to select or enter a file name, and save the file.

**CAUTION:** The block will not be appended to an already existing file; it will replace the file.

## LOADING A FILE

Clicking *Open*, or typing [Alternate] [O], displays the *item selector* dialog box. If you select an existing file, it will be loaded into the next available file space. If you enter a new file name, the next available file space will be opened for text input, and the file name will be displayed in the title bar.

When you *Open* a file, it is allotted a buffer equal to the length of the file plus 25,000 bytes. The buffer size is limited only by available memory. If your file is getting too large, *Save* it, and *Open* it again. You will have another 25,000 byte extension.

**CAUTION:** If you overflow the available buffer while entering text, an error will occur.

## NEW

*New*, or [Control] [L], opens a 25,000 byte buffer as the next available file space.

## LOADING A TEXT BLOCK

You can load a file into the text block buffer. Select *Load Block* from the *Block* menu, or type [Alternate] [L]. The Item Selector dialog box will appear. Select the file to load. It will be loaded into the block buffer. You can then use the *Paste Block* option from the *Block* menu, or type [Alternate] [P], to put the text where you want it.

**SEE:** *Cut and Paste*, page 3-7.

## CURSOR MOVEMENT

You can move the cursor on the screen using either the mouse or the arrow keys. To position the cursor with the mouse, place the arrow wherever you want the cursor to be, and press the left button. To move the cursor with the arrow keys, press the key whose arrow points the direction you want the cursor to move.

## SCROLLING

Scrolling from page to page and from one end of a line to the other can be accomplished with the mouse, or with the arrow keys in conjunction with the [Shift] key.

To scroll with the mouse, use the vertical slider bar for page-to-page movement, and the cursor for movement along a line of text. Select the area you want to view, then place the cursor with the mouse.

The shifted arrow keys move the cursor in this manner:

| | |
|---|---|
| [Shift] [Left-Arrow] | Beginning of the current line |
| [Shift] [Right-Arrow] | End of the current line |
| [Shift] [Up-Arrow] | Move to the previous page |
| [Shift] [Down-Arrow] | Move to the next page |

The cursor can be moved to the top of the text by selecting the *Top* option from the *Mark* menu, pressing the [Home] key, or by typing [Alternate] [T]. Move to the bottom of the text by selecting *Bottom* or by typing [Shift][Home] or [Alternate] [B].

## CUT AND PASTE

You can copy blocks of text, Move them from one place to another, or delete them altogether, using the *Mark*, *Erase* and *Paste* features of the *Block* menu.

## TO MARK A BLOCK OF TEXT

Place the cursor on the first line of the block and click *Mark Block* in the *Block* menu, type [Alternate] [M], or press the mouse button while holding down the [Shift] key. Place the cursor on the last line of the block and click *Mark Block*, type [Alternate] [M], or press the mouse button while holding down the [Shift] key. The selected block will be made bold-faced. This block will be the *current text block* until you select or load another one.

**NOTE:** You can only mark full lines of text. You can't split a line for block operations.

## TO UNDO BLOCK SELECTION

Click on the *Hide Block* option of the *Block* menu, or type [Alternate] [H]. The text will return to the normal font on the screen.

**TO PASTE A BLOCK OF TEXT**
Mark the block and place the cursor on the line where you want to put the text and click *Paste Block* in the *Block* menu, or type [Alternate] [P]. The current text block will be placed in the new location, starting at the beginning of the line containing the cursor.

**NOTE:** You can only paste starting at the beginning of a line. You can't insert text into the middle of a line.

**TO ERASE A BLOCK OF TEXT**
Mark the block and select *Erase Block* from the *Block* menu, or type [Alternate] [E]. The block will be removed from the screen, but it will be retained as the current block.

**TO COPY A BLOCK OF TEXT**
Mark it, then paste it where you want it.

**TO MOVE A BLOCK OF TEXT**
Since an erased block stays current until you mark another one, you can move a block by erasing it and then pasting it where you want.

## FIND AND REPLACE

The *Find* menu has options to allow you to find a particular string and, if you want, replace it. You can either replace the next occurrence of a string or replace all occurrences of it. Searching can be case-sensitive or not, as you desire.

To find a string, first select *Find What* from the *Find* menu, or type [Alternate] [F]. Enter the string you want found on the target line in the dialog box. If you are going to replace the string, enter the new string on the replacement line. You can either match or ignore the target string's case by clicking the appropriate button. Click the *OK* button or press [Return] to enter the Find and Replace information.

After entering the *Find What* information, you can search upwards or downwards from the current cursor position by clicking *Find Previous* or *Find Next* in the *Find* menu. You can also type [Alternate] [U], for upwards, or [Alternate][D], for downwards. The search criteria will remain set until you use *Find What* to change them.

When you have found the string you want, you can replace it with the target string clicking *Replace Next* in the *Find* menu, or by typing [Alternate] [R].

To replace all occurrences of the target string, click *Replace All* in the menu. There is no equivalent keyboard command.

## POSITION MARKERS

You can place markers on up to 4 lines so you can move the cursor to them with a single menu click or keystroke.

To mark a position, place the cursor on the line you want, and click *Set* in the *Marker* menu to select a marker.

To return to a marker, click its *Goto* option in the *Mark* menu, or press a function key, 1 through 4.

You can reset a marker at any time by placing the cursor in a new position and clicking the appropriate *Set* option in the *Mark* menu.

**CAUTION:** Markers are set by *LINE NUMBER* and apply in all 3 text buffers.

## PRINTING

To print the currently displayed file, click *Print File* in the *Options* menu.

To print a text block, first mark the block, then click *Print Block* in the *Block* menu.
**SEE:** *Cut and Paste*, page 3-7.

# EDITOR COMMAND SUMMARY

Many of the Editor's functions can be invoked my both menu selections and keyboard commands. The list below is a summary of menu options and their keystroke equivalents.

**FILE MENU**
**SEE:** page 3-12

**BLOCK MENU**
**SEE:** page 3-14

**FIND MENU**
**SEE:** page 3-16

**MARK MENU**
**SEE:** page 3-17

**OPTIONS MENU**
**SEE:** page 3-18

**KEYSTROKE SUMMARY**
**SEE:** page 3-19

# THE FILE MENU

**NEW** **[Alternate] [N]**
Opens a new 25,000 byte buffer in the currently displayed file space. The file in the currently displayed space is cleared and is not saved.

**OPEN** **[Alternate] [O]**
Loads a file or creates a new one in the next available file space. Presents the *Item Selector* dialog box. Sets the buffer to file length plus 25,000 bytes.

**SAVE FILE** **[Alternate] [S]**
Saves the currently displayed file under its current name. There is no prompting dialog box except on the first *Save* after *New.*

**SAVE AS** **No key equivalent**
Saves the currently displayed file under a specified name. The current file name is the default. Presents the *Item Selector* dialog box. The default file name is the last name used for a save.

**QUIT** **[Alternate] [Q]**
Leaves the editor. If any text has been changed since the last *Save,* displays a warning so you can Save! if you want.

**COMPILER** **No key equivalent**
Saves the current text buffers under their current names and runs the compiler. Does not prompt before saving.

**LINKER** **No key equivalent**
Saves the current files under their current names and runs the linker. Does not prompt before saving.

# BLOCK MENU

**MARK BLOCK [Alternate] [M]**
Marks lines of text for cutting and pasting. You must mark the beginning and end of a block. The block is presented in bold-face on the screen when the second mark is placed. Marks whole lines only.

**NOTE:** Holding the [Shift] key while clicking the mouse button also performs *Mark Block*. It is the most convenient method.

**ERASE BLOCK [Alternate] [E]**
Removes a marked block from the file space. The block remains in the paste buffer until another block is marked.

**PASTE BLOCK [Alternate] [P]**
Inserts the *current text block* into the displayed text at the beginning of the line containing the cursor. You can't insert text into the middle of a line.

**HIDE BLOCK [Alternate] [H]**
Restores a selected, bold-faced block to the normal screen font. The current text block is not changed.

**LOAD BLOCK No key equivalent**
Loads a file into the paste buffer. The paste buffe becomes the current text block. Presents the *Item* Selector *dialog box*

**SAVE BLOCK No key equivalent**
Saves the current text block to a file. Presents the *Item Selector* dialog box.

**PRINT BLOCK No key equivalent**
Sends the current text block to the printer.

# FIND MENU

**FIND WHAT** **[Alternate] [F]**
Presents a dialog box containing the target and replacement strings for search and replace functions. Does not initiate search or replace.

**FIND PREVIOUS** **[Alternate] [U]**
Initiates a search from the current cursor position towards the beginning of text. Uses the target string in the *Find What* dialog box.

**FIND NEXT** **[Alternate] [D]**
Initiates a search from the current cursor position towards the end of text. Uses the target string in the *Find What* dialog box.

**REPLACE NEXT** **[Alternate] [R]**
Use after search to replace the target string at the current cursor position with the replacement string specified in the *Find What* dialog box.

**REPLACE ALL** **No key equivalent**
Replaces every occurrence of the target string in the displayed file with the replacement string specified in *Find What.* Prompts for replace with query or replace without query.

# MARK MENU

**Set Mark 1** **No key equivalent.**
**Set Mark 2** **No key equivalent.**
**Set Mark 3** **No key equivalent.**
**Set Mark 4** **No key equivalent.**

*Set* puts place markers in the displayed text, then when one of these menu items is selected or a function key is pressed, the cursor will be moved to the marked line.

**F1 Go To Mark 1** **[F1]**
**F1 Go To Mark 2** **[F2]**
**F1 Go To Mark 3** **[F3]**
**F1 Go To Mark 4** **[F4]**

Clicking one of these options or pressing the equivalent Function Key moves the cursor to the appropriate marked text line.

**TOP** **[Alternate] [T] or [Home]**
Moves the cursor to the first line of text.

**BOTTOM** **[Alternate] [B] or [Shift][Home]**
Moves the cursor to the last line of text.

# OPTIONS MENU

**GOTO LINE** **[Alternate] [G]**
Presents a dialog box asking for a line number. Moves the cursor to the line number you enter.

**NOTE:** By using one buffer for a your Pascal source file and another for the Compiler's *.ERR* error file, you can find and correct errors easily.

**TABSIZE** **No key equivalent**
Presents a dialog box containing the current tab interval value. Sets the tab interval to the value you enter.

**INSERT** **[Insert] (Toggles mode)**
Selects insert text entry mode. Text under the cursor will be preserved and pushed to the right as you type.

**OVERWRITE** **[Insert] (Toggles mode )**
Selects overwrite text entry mode. Text under the cursor will be replaced by what you type.

**AUTO INDENT** **No key equivalent**
Selects Automatic indentation. When you press [Return], the cursor will be placed at the same tabbed margin as the preceding line. To change the tabbed margin of the current line, use [Tab] or [Backspace].

**PRINT FILE** **No key equivalent**
Sends the displayed file to the printer.

# KEYSTROKE SUMMARY

## WORDSTAR COMMANDS

The following Wordstar(tm) keystrokes are available as Editor comands:

| | |
|---|---|
| **[Control] [S]** | Cursor Left 1 Character |
| **[Control] [D]** | Cursor Right 1 Character |
| **[Control] [E]** | Cursor Up 1 Line |
| **[Control] [X]** | Cursor Down 1 Line |
| **[Control] [R]** | Cursor Page Up |
| **[Control] [C]** | Cursor page Down |
| **[Control] [ I ]** | Tab |

## CURSOR CONTROL

The cursor can be controlled from the keyboard using the arrow keys:

| | |
|---|---|
| **[Left Arrow]** | Cursor Left 1 Character |
| **[Shift] [Left Arrow]** | Beginning of Text Line |
| **]Right Arrow]** | Cursor Right 1 Character |
| **[Shift] [Right Arrow]** | End of Text Line |
| **[Up Arrow]** | Cursor Up 1 Line |
| **[Shift] [Up Arrow]** | Page Up |
| **[Down Arrow]** | Cursor Down 1 Line |
| **[Shift] [Down Arrow]** | Page Down |

## TEXT FUNCTIONS

These text functions are implemented in the Editor:

| | |
|---|---|
| **[Shift] [Delete]** | Delete the current text line. |
| **[Tab]** | Move the cursor to the next tab position. |
| **[Insert ]** | Toggle Insert/Replace text mode. |

# THE COMPILER

The Personal Pascal compiler changes source programs with the *.PAS* extender into a form that can be linked with other files. When the Linker is through, your program can be run from the Personal Pascal Manager using the *Run Program* item in the *Files* menu, or from the desktop.

If your program compiles without errors, the Compiler generates a *FILENAME.O* file. ( *FILENAME* is the name of the file you are compiling. ) This is the file to link in order to produce an executable program.

If your program contains something that the compiler is unable to translate, the compiler will produce an error message or put a list of the errors it finds into a file with the extension *.ERR*, depending upon your choice of options. The *.ERR* file goes into the same folder as the file being compiled and has the same name as your *.PAS* source file. There is a list of all the error numbers and messages the compiler produces in the Appendix.

**COMPILING A PROGRAM**

You can run the Personal Pascal compiler from two places: the Manager and the Editor. To compile a program when you are at the Manager level, move the mouse over the *Files* menu title and the *Files* menu will appear. Move to the *Compile* or *Compile All* item and click the mouse. A dialog box will appear and you can choose the file you want to compile.

# COMPILER OPTIONS

Compiler options are selected from the Manager. If you move the mouse over the Manager's *Options* title, you will see that one of the items available is *Complier Options*. If you click on this item, the *Compiler Options* dialog box will appear. The options in this dialog give you extensive control over the Personal Pascal compiler.

For convenience, the *Compiler Options* discussion from the section on the Compiler is repeated here:

**PROGRAM TYPE**
You can compile a GEM program, an ACCessory, or a TOS program ( no GEM functions ). Click the applicable radio button for the type of program you are compiling. If your program is not an accessory and uses any of the GEM/Pascal library functions, you should leave the GEM option selected. Accessories assume the use of the GEM/Pascal library.

**NOTE:** There are a few special considerations when writing desk accessories.
**SEE:** *Writing Desk Accessories*, in the appendix.

## PAUSE AFTER ERRORS

When the compiler encounters an error, it puts an identifying error number and message into a file, using the name of the file being compiled, but appending the extension *.ERR*. If *Pause After Errors* is selected, an alert box will appear on the screen, asking whether you want to continue the compilation, cancel it and return to the Manager, or cancel and go directly to the Editor. If you choose to cancel and go to the editor, your file will be reloaded into the first buffer and you will see the cursor marking the the error's position, and an error message. When you correct the error, you can run the Compiler directly from the Editor.

## CHAIN TO LINKER

When you check Chain to Linker, the compiler will automatically run the linker when your program is successfully compiled. The linker will use the program type you have specified, GEM, TOS, or ACC, to link the appropriate Personal Pascal files to your program.

## FULL DEBUG MODE

When the Full Debug Mode box is checked, the compiler will include code to support debugging. If there is an error while your program is running, its execution will stop and you will be given information about the error, including:

**An error description**
**The subprogram name**
**The line containing the error**
**The current Program Counter value.**

**NOTE:** If an error occurs in the operating system, the subprogram name *Unknown* will be displayed.

## CLEAR VARIABLES

When Clear Variables is checked, the compiler generates code to clear local variables, function return values, and space returned by the *New* procedure.

## STACK AND POINTER CHECKING

When the Stack and Pointer Checking option is checked, the compiler generates code to check the stack and pointers for overflow at run-time. If an error

occurs while your program is running, its execution will stop and an alert box or a text line ( for TOS programs ) will appear, describing the error. If you are in full debug mode as well, you will be given additional information about the error.
**SEE:** *Full Debug Mode*,

**RANGE CHECKING**
When the Range checking option is checked, the compiler will generate code to check subranges and array indices at run-time. If a range error occurs while your progam is running, execution will stop and an alert box or a text line ( for TOS programs ) will appear, describing the error. If you are in full debug mode as well, you will be given additional information about the error.
**SEE:** *Full Debug Mode*, page 4-4.

**PRINT LISTING**
When this option is checked, the compiler prints a listing to the screen as it compiles your program. This option is *UNCHECKED* as default.

# COMPILER DIRECTIVES

The Personal Pascal compiler accepts a number of compiler directives. These directives are embedded in the source as comments. Comments starting with a dollar sign {$ } are compiler directives.

The directives are made up of a single letter followed by a plus or minus sign to switch the directive on or off. One comment can contain more than one directive if they are separated by commas.

**NOTE:** There are 2 variations of the Include directive, {$I}. One of them is followed by a file name, instead of plus or minus. The use of the {$I} directive with a file name is the only case in which a space may be present within a directive.

A compiler directive can be switched on or off anywhere in the source, unless expressly forbidden in its description. When a directive is set or reset, it may be restored to its former state by specifying it with an equal sign e.g: {$P=}.

**EXAMPLES:**

```
{$R-,T-,M-,P+}
(*$R+*)
{$I incfile}
{$P-}
     ...some intervening code...
{$P=}
```

All of the compiler directives available in Personal Pascal are explained in the following pages.

**{$C+} CLEAR** **{$C-} = OFF**
The Clear directive causes the compiler to generate code to clear:

**Local variables after invoking a subprogram**
**Dynamic memory returned by *New***
**Initial function return values**

**NOTE:** You can invoke this option from the Clear variables box in the Compiler Options dialog.

**{$D+} DEBUG** **{$D-} = OFF**
This directive causes the compiler to include source line number and subprogram names in the generated code. Run-time errors are then reported by subprogram name and source line number.

**NOTE:** This option may only appear at the beginning of a program.

**NOTE ALSO:** You can invoke this option from the *Full Debug* mode box in the *Compiler Options* dialog.

**{$E+} EXTERNAL ACCESS** **{$E-} = OFF**
The External Access directive controls whether the compiler makes subprograms accessible to other modules.
**SEE:** *Modular Compilation*, page 4-11, and Appendix.

**NOTE:** *GEMDOS, BIOS,* and *XBIOS* subprograms can't be accessed by other modules, but they may be re-declared in each module without conflict.

**{$I FILENAME} - INCLUDE {$I-} = OFF**
The Include directive is used to include another file in the source at compile time. After reading the include file, the compiler continues to compile the original file. The file name may contain a path name and an extension. The extension defaults to *.PAS*.

**NOTE:** Nested include files are not allowed.

**B{$I+} - LONG INTEGER B{$I-} = OFF**
The Long Integer directive is used to set all *Integer* references to *Long integer* references. {$L-} makes *Integer* references *Short_Integer* again.

**{$M} - MODULAR COMPILATION**
The Module directive causes the compiler to compile a module rather than a main program. When compiling a module, the main program should be empty because it never can be executed.
**SEE:** *Modular Compilation* page 4-11.

**NOTE:** This option may only appear at the beginning of a program module.

**{$P} - POINTER RANGE CHECKING**
The Pointer Checking directive causes the compiler to generate code to check pointer ranges before using them. Pointers are compared to the bounds of the run-time heap for validity.

**NOTE:** This option must be switched *off* when pointers are used to access the base page or addresses which are outside the heap.

**NOTE ALSO:** You can invoke this option from the *Range checking* box in the *Compiler Options* dialog.

**{$R} - RANGE CHECKING**
The Range Checking directive controls whether the compiler generates code to check subrange and array bounds. It also controls whether code is generated to check the bounds of strings and string parameters.

**{$Snumber} - SPACE FOR STACK AND HEAP**
**EXAMPLE:** {$S20} reserves 20 Kbytes.
This directive tells the compiler how much space to allocate for the stack and heap, (in Kbytes) and may only appear at the beginning of a program. This directive is useful when compiling for TOS because all available memory is allocated to the Stack/Heap space (because you don't need GEM space).
**SEE:** ${U} - *User Memory directive* page 4-10.

**NOTE:** You may not use both the {$S} and {$U} compiler directives at the same time.

**{$T} - TEMPORARY SPACE CHECKING**
The Temporary Space Checking directive tells the compiler to check the stack against the heap for overlap, and to generate code at the start of each subprogram to make this check.

**{$Unumber} - USER MEMORY**

**EXAMPLE:** {$U20} reserves 20 Kbytes.

This directive lets you specify how much memory you want left over for the system after the Stack/Heap space is allocated. (in Kbytes) This is useful when compiling for GEM if you have a large resource. The default is {$U10} when compiling for GEM, and this is enough in most cases, though complex programs may use 20K or more.

**NOTE:** If you are chaining programs, you must use either the {$S} or {$U} directive to allow enough space for the chained program to load.

**NOTE ALSO:** You may not use both the {$S} and {$U} compiler directives at the same time.

# MODULAR COMPILATION

Personal Pascal extends standard Pascal by allowing you to compile program modules separately and link them later. When developing a large program, you can write and debug separate modules and then link them, reducing the time you spend compiling. You can also use modular compilation to create libraries of subprograms that you use frequently.

## WRITING A MODULE

A module is specified by switching on the *Module* directive {$M+} in the first line of a program (before the PROGRAM declaration). The module can contain as many subprograms as you wish, but the main body must be empty. Also, the *External Access* directive must be on ({$E+}) when you declare a subprogram that you want to be accessible outside the module.

If the subprograms in the module do not use the main program's variables, the module need contain only those declarations and definitions required for the subprograms' parameters. If the subprograms do use the main program's variables, the module must contain ALL global definitions (TYPE and CONST) and variable declarations in *EXACTLY THE SAME ORDER* as they appear in the main program. You can do this easily by creating a file that contains these definitions and declarations, and then using the {$I} directive to include it in both the main program and the module.

**NOTE:** If you change *any* of these definitions or declarations, you must recompile *all* modules and programs that use them. This makes the *Compile All* command very useful.

If you compile the main program with the *Full Debug Mode* box checked, you must also compile all other modules with it checked. If you don't, strange effects may result if a run-time error occurs.

# THE LINKER

The Personal Pascal linker connects your compiled Pascal program to other linkable files so that your program can be run. You can run it either from the Personal Pascal Manager, using the *"Run Program"* item in the *Files* menu, or from the desktop by double-clicking its icon.

You will link different files to your program, depending upon what you are doing. If you are not using link files or libraries of your own, you still must link your compiled program to the Pascal Run-time Library and the GEM/Pascal Library ( if you're linking for GEM ) to make it self-sufficient.

**LINKING A COMPILED PROGRAM**

You may run the Personal Pascal linker from two places: the Manager and the Compiler.

To link a program from Manager, select the *"Link"* item from the *"Files"* item. The *"Item Selector"* will appear and you can choose the file you want to link.

To link a program immediately after compiling it, check the *"Chain to linker"* box in the Compiler Options dialog. If your program compiles without errors, the linker will load and link your program automatically.

# LINKER OPTIONS

Linker options are selected from the Manager. If you move the mouse over the Manager's Options title, you will see that one of the items available is *"Linker Options"*. If you click on this item, the Linker Options dialog box will appear. The options in this dialog give you extensive control over the Personal Pascal linker. For convenience, the *"Linker Options"* discussion from the section on the Manager is repeated here:

**PROGRAM TYPE**
As with the Compiler, you can tell the Linker to link the appropriate GEM/Pascal files for a GEM program, a TOS program, ( no GEM functions ) or an ACCessory. Click the applicable radio button for the type of program you are linking. If your program uses any of the GEM/Pascal library functions, you should leave the GEM option selected. Accessories assume the use of the GEM/Pascal library.

**NOTE:** There are a few special considerations when writing desk accessories.
**SEE:** *"Writing Desk Accessories"*, in the appendix.

## ADDITIONAL LINK FILES

The linker automatically links Pascal object files and Pascal library files to produce your program. If you used the Modular compiler directive {$M}, you can enter the names of additional files that you want linked, such as assembly language modules, or other Pascal object files. You can enter as many file names, separated by commas, as will fit on the lines given.

**NOTE:** There is one restriction; a filename may not extend from the first line onto the second line.

## ASK FOR FILE TO LINK

When this box is checked, the Linker prompts you for the name of the file you want linked. The default filename is the last file edited, compiled or linked in the current session.

## IMPORTANT NOTE ABOUT LINK FILES:

The Personal Pascal version 2 linker works with files in 2 formats: its own unique format, and the DRI link format which is the Atari standard. There is a utility available from OSS to convert Personal Pascal format link files to the DRI standard.

**SEE:** *Assembly Language*, page 6-142.

BLANK
PAGE

# LANGUAGE REFERENCE

This Section of the Personal Pascal manual contains a definition and discussion of Pascal as implemented by version 2 of Personal Pascal. This is most definitely a reference manual and not a tutorial, so we must point you to the books mentioned at the beginning of this manual if you need help learning to program. Generally, if you restrict your programming to those capabilities described in this section and ignore the GEM interface described in Section 7, you will find that Personal Pascal version 2 will correctly compile and execute example and problem programs found in tutorials and other similar books. You should compile and link your programs using the TOS options.

This part of the manual explains:

**Pascal Definitions**
**Pascal Program Format**
**Special Topics**
**Predefined Pascal Subprograms.**

## PREFACE

For the most part, Personal Pascal conforms to standard Pascal as that language was formulated by the International Standards Organization (ISO). In order to make Personal Pascal more viable in the interactive environment of ST microcomputers, however, OSS has implemented several extensions to the ISO standard. The side effect has been that there may be a very few areas where Personal Pascal is unable to accept ISO standard programs. For this we apologize, but we feel that on the whole the language is much the stronger for the changes. Certainly Personal Pascal version 1 quickly became a *de facto* standard for Pascal programming on the ST. We firmly believe that version 2 will be an even better and more successful product.

The most useful Personal Pascal extensions to the ISO standard include:

**Random Access Files**
**STRING Data Type**
**OTHERWISE option in CASE statements**
**A Generic LOOP Statement**
**Two sizes of integer variables and constants**
**Optional Hexadecimal notation for integers**
**Modular Compilation**
**Flexible ordering of Declarations.**
**Assembler, System, and C calls**

## OVERVIEW

There are four major parts to this section of the Personal Pascal manual: Definitions, Pascal Program Format, Special Topics, and Predefined Subprograms.

Strictly speaking, the Personal Pascal language is described in the PASCAL PROGRAM FORMAT part. That part alone gives a moderately formal presentation of the language, including syntax diagrams.

If we had tried to take a formal approach to each Pascal programming topic, this manual would be three times the size it is now. Instead, we begin with a DEFINITIONS section containing less formal definitions of the basic building blocks of Pascal. We have also given PREDEFINED SUBPROGRAMS their own section, even though many of these subprograms are part of ISO standard Pascal.

Finally, we have included a section for SPECIAL TOPICS which discusses features of Personal Pascal not found in standard Pascal and advanced usages that did not fit easily into any other categories.

# DEFINITIONS

Most algorithmic computer languages, for example, Pascal, C, BASIC, Fortran, Algol and Ada, share basic concepts such as variables, constants, comments, arithmetic operators and expressions. This section explains how Personal Pascal treats these language elements.

Naturally, every language has its quirks and variations, and Personal Pascal is no exception. As you read these Definitions, be alert to the differences between Personal Pascal and whatever computer languages you may already know or be studying.

More importantly, the definitions made in this sub-section are used throughout the rest of this manual. They are usually not explained further as they are used, so if you do not have a firm grasp on these definitions, you will find yourself constantly turning back to this section as you program.

This section is divided into two major topics: Pascal Elements and Expressions. The first is obvious: we need to define and describe the elements of Pascal. The second is necessary: Computer programming relies heavily on an ability to "express" a formula or relationship in terms of underlying building blocks.

**ELEMENTS** page 6-5

**EXPRESSIONS** page 6-23

# PASCAL ELEMENTS

In one sense, every Pascal program can be broken down into the individual characters that the programmer enters in an effort to produce something "acceptable" to the language compiler. For example, a number such as 375 may be thought of as being composed of the characters '3', '7', and '5'. But it is not very productive (and pretty darned silly) to constantly break programs down this far. Instead, we will here define the elements that the rest of this manual treats as the fundamental building blocks of Personal Pascal.

In a few cases, you could argue that we have *arbitrarily* designated something as "fundamental". For example, strings are quite clearly a special case of groups of characters. Nonetheless we think these 6 elements represent a good working list:

## SPECIAL SYMBOLS

Personal Pascal supports these special symbols:

```
+  -  *  /  =  <  >  (  )  [  ]

{  }  .  ,  :  ;  '  ^  |  &  ~

   :=    <>    <=    >=    ..
```

In general, these familiar symbols mean something in Pascal that is similar to what we usually associate with them, but remember that they do not act exactly as we would expect them to in a mathematics formula or an english sentence. Also, please notice that the symbols in the last row are actually made up of two characters each.

Some of the character symbols above have alternate forms left over from the dark ages when computers didn't have keys like { or [. These alternates are:

```
(*    for    {

*)    for    }

(.    for    [

.)    for    ]

@     for    ^
```

Personal Pascal also recognizes the words in this list:

| | | |
|---|---|---|
| **AND** | **ARRAY** | **BEGIN** |
| **BIOS** | **C** | **CASE** |
| **CONST** | **DIV** | **DO** |
| **DOWNTO** | **ELSE** | **END** |
| **EXIT** | **EXTERNAL** | **FILE** |
| **FOR** | **FORWARD** | **FUNCTION** |
| **GEMDOS** | **GOTO** | **IF** |
| **IN** | **LABEL** | **LOOP** |
| **MOD** | **NOT** | **OF** |
| **OR** | **OTHERWISE** | **PACKED** |
| **PROCEDURE** | **PROGRAM** | **RECORD** |
| **REPEAT** | **SET** | **THEN** |
| **TO** | **TYPE** | **UNTIL** |
| **VAR** | **WHILE** | **WITH** |
| **XBIOS** | | |

These word symbols are often called reserved words or keywords because they make up Pascal's basic vocabulary and can't be redefined within a program.

## IDENTIFIERS

There are two kinds of identifiers: Those you define yourself ( or find in system libraries ) and predefined identifiers.

## PROGRAMMER DEFINED IDENTIFIERS

Identifiers allow you to add to Pascal's vocabulary. When you define or declare something within a Pascal program, you give it a name, or identifier, so that you can later refer to it without confusing Pascal.

All identifiers must begin with a letter, but after that may contain letters, digits, or the underscore character ( _ ):

NOTE: Personal Pascal makes no distinction between lower and upper case letters. "ABC" and "abc" are the same.

Practically speaking, there is to limit to the number of characters you can have in an identifier used internally to a Pascal program, although the real limit is somewhere between 100 and 120 characters. External identifiers have smaller limits, however. Pascal externals have a maximum length of 8 characters. C language externals have a maximum length of 7 characters.

## PREDEFINED IDENTIFIERS

Personal Pascal has several predefined identifiers. They are described more fully later in the Reference manual:

### PREDEFINED DATA TYPES:

| | |
|---|---|
| **Alfa** | **Byte** |
| **Boolean** | **Char** |
| **Integer** | **Long_Integer** |
| **Real** | **Short_Integer** |
| **String** | **Text** |

### PREDEFINED CONSTANTS:

| | |
|---|---|
| **False** | **Input** |
| **Long_Maxint** | **Maxint** |
| **Nil** | **Output** |
| **True** | |

## PREDEFINED PROCEDURES:

| | | |
|---|---|---|
| BasePage | Chain | Cmd_GetArg |
| Delete | Dispose | Erase |
| Get | Halt | Insert |
| IO_Check | IO_Result | Mark |
| Message | New | Pack |
| Page | Put | Read |
| Readln | Release | ReName |
| ReSet | ReWrite | UnPack |
| Write | Writeln | |

## PREDEFINED FUNCTIONS:

| | | |
|---|---|---|
| Abs | ArcTan | Chr |
| Clock | Close | Cmd_Args |
| Concat | Copy | Cos |
| Eof | Eoln | Exp |
| Filename | Handle | KeyPress |
| Length | Ln | Long_Round |
| Long_Trunc | MemAvail | Odd |
| Option | Ord | Pos |
| Pred | PwrOfTen | Round |
| Shl | Short_Round | Short_Trunc |
| Shr | Sin | SizeOf |
| Sqr | Sqrt | Succ |
| Trunc | | |

## LABELS

Labels are very specialized and rarely used in Pascal. They are used only when non-linear program flow can't be avoided in a program, and they require the dreaded GOTO statement to be useful. Since a major point of structured programming is to avoid such indiscretions, this is a rare occurrence indeed.

A label is a decimal digit sequence. The range of labels in Personal Pascal is 0 to 32767. These are valid labels:

```
LABEL 10 ;
LABEL 1 ;
LABEL 3543 ;
```

Labels are used only in GOTO statements, such as

```
GOTO 10 ;
```

**SEE:** *GOTO statements*, page 6-108
*Label Declarations*, page 6-43

## CONSTANTS

Pascal understands four types of constants: numbers, characters, logical states and strings.

Because there are different types of numbers, Pascal breaks the data type NUMBER into three separate data types, making six classes of constants to be discussed here. Each of these classes corresponds directly to one of Personal Pascal's fundamental predefined data types. Further implications of these types are described in the section titled TYPES. One consequence is that we tend to make little distinction between constant data and variables of the same types. The type classes are:

**Strings**
**Real**
**Short_Integer**
**Long_Integer**
**Boolean**
**Char**

The last four of these are ordinal types - the set of data they represent is finite and ordered. Real data is not ordinal because its data set is not finite. Although the set of all Pascal strings is finite, it is not considered to be ordered, and so strings are not ordinal.

## CHARACTER STRINGS

A character string is one or more printable characters enclosed within single quotes ('). If you want to use a single quote within a character string, use two single quotes ('').

All of the following are valid character strings:

'this is a string'

'5 238 test 58 '' single quote image'

**NOTE:** A single character between quotes is usually considered to be a character constant, not a string.

**ALSO:** No string may exceed *255 characters* in length.

## REALS

Although Real is not an ordinal type, it does have order since 1.0 is less than 1.1. You can think of Real values in Pascal as close approximations of real values in mathematics but, due to limits imposed by internal representation, Pascal Real values are a subset of mathematical reals.

Real numbers may be represented only in decimal notation, and are made up of two parts: a mantissa and a scale factor (exponent). The mantissa is either a decimal integer or a floating point decimal number. The scale factor is a decimal integer prefixed by an "e". This e means "times ten to the power". For example, 2.5e3 is the same as 2.5 times 10 to the third power (1000), which equals 2,500. Unless a real number has a scale factor, the mantissa must be a decimal number with at least 1 digit on each side of the decimal point. 1e20, 535.0, and 0.4348E-12 are examples of valid reals.

In Personal Pascal the mantissa has 11 digits of precision maximum, and the scale factor (exponent) has a range -38..38. This means that the largest Real is about 1.0E38, and the smallest (closest to zero) is about 1.0E-38.

## ORDINALS:

### INTEGERS

Decimal integers may be preceded with a plus (+) or minus (-) to denote sign, and consist of decimal digits ('0'..'9'). Personal Pascal assumes that a decimal integer is positive (+) if you do not specify its sign.

100, -4323, and 0 are examples of decimal integers.

Hexadecimal integers must be prefixed with a dollar sign ($) or a double quote ("), and consist of hexadecimal digits '0'..'9' and 'A'..'F'. The digits 'a'..'f' are also allowed. These are examples of valid hex integers:

"64E $AFFE $affe

### SHORT_INTEGERS

Short_Integer values are whole numbers ranging between -32,767 and +32,767 decimal. This number, 32,767 is the value of the predefined constant Maxint.

Hexadecimal Short_Integer values range between $0 and $FFFF.

Some examples of valid Short_Integer values are:

0 31000 -743 +2957 $6FFE "1000 $7fff

## LONG_INTEGERS

Long_Integer values range between -2,147,483,647 and +2,147,483,647 decimal. This number is the predefined constant Long_Maxint. Hexadecimal Long_Integer values range between $0 and $FFFFFFFF.

Some examples of legal Long_Integer values are:

1000000 -58104 $6a479e "00FFFF00

**NOTE:** Because integers are *signed* numbers, while their *hex* representation is *not,* seemingly positive hex values may be negative integers.

Short_Integer values:

$8000 - $FFFF

are negative, as well as Long_Integer values:

$80000000 - $FFFFFFFF

If the most significant bit is 1, the integer value is negative.

**CAUTION:** When a Short_Integer value is used where a Long_Integer is required, Personal Pascal automatically extends the Short_Integer, *including its sign.* Thus, $FFFF is extended to $FFFFFFFF.

## CHARACTERS (CHAR)

The value of CHAR data consists of a single ASCII character. Some examples of valid CHAR values are:

'a' 'z' ' ;'

You can assign non-printable characters to CHAR variables using the CHR transfer function, or using the special Personal Pascal notation: #n.

**EXAMPLES:**

#48 Is the same as '0'
#13 Is The ASCII 'Return' character

**NOTE:** Character constants from #0 to #255 are legal uses of this notation.

## BOOLEAN

This data type has only two values, represented by the predefined words *FALSE* and *TRUE*. For purposes of comparison, *FALSE* is less than *TRUE*.

## LANGUAGE DIRECTIVES

Language directives are different from compiler directives. Compiler directives tell the compiler to perform special tasks such as listing, code generation, and the like. Language directives notify the compiler of special situations within your Pascal program.

Personal Pascal supports the ISO standard language directive FORWARD as well as EXTERNAL, C, GEMDOS, BIOS and XBIOS directives, all used to interface to program elements outside the scope of standard Pascal, such as separately compiled modules, other languages and the ST operating system.

Language directives ALWAYS are used after a PROCEDURE or FUNCTION header, to denote that the named routine is not followed by the normal Pascal declarations or body.
**SEE:** *Subprogram Declarations,* page 6-84

## FORWARD

The FORWARD directive allows mutually recursive subprograms (A calls B which calls A). When you use the FORWARD directive in a subprogram heading, you must fully declare that subprogram somewhere later in the program. When you do, you need only specify whether it's a procedure or function and its identifier. Redeclaration of its formal parameters is prohibited. An example of a FORWARD declaration will clarify is use and usefulness:

```
PROCEDURE First(i,j:Integer) ;
  FORWARD ;

PROCEDURE Second(k,l:Integer) ;
  BEGIN
    ...
    First(k,l) ; { calling First }
    ...
  END ; { end of Second }

PROCEDURE First ;
  { now comes the declaration of what First does.
    Notice that no parameters are specified. }
  BEGIN
  ...
  Second(i,j) ; { calling Second, which calls First }
  ...
  END ; { end of First }
```

## EXTERNAL

The EXTERNAL directive allows you to reference subprograms that are in different modules. EXTERNAL assumes that the subprogram was written either in assembly language ( using the Personal Pascal register, parameter, and return value conventions described in the section on *Special Topics* ) or in Personal Pascal and compiled with the {M+} modular compilation compiler directive.

**CAUTION:** In contrast to most Pascal identifiers, *EXTERNAL* identifiers *ARE* case sensitive.
**SEE:** *Modular Compilation* page 4-11

## C

The C directive allows you to reference subprograms written and compiled using Digital Research's C compiler, or assembly language programs that were written using the same linking scheme as Digital Research's C.

**NOTE:** Personal Pascal converts the subprogram name to lower case and precedes it with an underscore character to meet C conventions.

## GEMDOS, BIOS, XBIOS

These directives allow you to make direct system calls to the GEMDOS, BIOS, and XBIOS levels of TOS. Each directive takes one integer constant as a parameter, defining which GEMDOS, BIOS or XBIOS routine you want to access.

As an example, TOS provides XBIOS call number 17 which returns a 24-bit random number each time it is called. From Personal Pascal, the following function declaration is sufficient to provide access to that call:

```
FUNCTION Random : Long_Integer ;
            XBIOS( 17 ) ;
```

Then, in your program, you may may request random numbers as easily as this:

```
IF (Random & 3) = 0 THEN
   { this code will be executed 25% of the time }
   ...
```

**NOTE:** Further examples of these directives are to be found in the demo programs on your Personal Pascal diskettes.

## COMMENTS

Comments allow you to describe what you're doing in a Pascal program without getting syntax errors because you used English instead of Pascal. Comments begin with either a left brace, {, or its alternate, (*, and are terminated by the matching terminator, } or *). Between these symbols you can have any number of text lines and the compiler will ignore them.

NOTE: Comments that begin with '{' must be terminated with '}', and ones that begin with '(*' must end with '*)'. This matching allows you to nest comments in Personal Pascal, e.g.:

(* { A comment } within a comment *)

**CAUTION:** Any comment beginning with {$ will be interpreted as a COMPILER DIRECTIVE, not to be confused with LANGUAGE DIRECTIVES as described above.
**SEE:** *Compiler Directives* page 4-6.

# EXPRESSIONS

Many places in pascal you will find that the syntax requires a *value.* The only values we have described so far are *constants. Variables* and *Functions* also have associated values, but a value in Pascal is not limited to a single such element; it can be expressed as a formula. These formulas are called *expressions* and are made up of operands representing values and operators that manipulate the values.

# EXPRESSIONS
# ORDER OF PRECEDENCE

Because operators can introduce ambiguity into an expression, a standard order of operator evaluation is needed. This order is called *precedence*, and is:

**First:** **NOT ~**
{negation operators}

**Second:** *** / DIV MOD AND &**
{multiplication operators}

**Third:** **+ - OR |**
{addition operators}

**Last:** **= <> > >= < <= IN**
{relation operators}

Operators on a row have left to right precedence within the row. Using this precedence table, we see that 8*4-7, for example, would be evaluated as (8*4)-7 because the "*" operator has higher precedence than the "-", giving us a final result of 25.

Please keep in mind that the terms *negation, multiplication, addition,* and *relation* do not imply any specific data type. They merely describe the kind of action an operator performs on a compatible data type.

# OPERATORS, OPERANDS AND RESULT TYPES

If an expression consists of a single operand (i.e. has no operators) the resulting value is obviously of the same data type as the operand. When expressions become more complex, their results can take one of two forms: numeric or boolean. The form can be determined from the last operator evaluated. The tables on the following pages show the data type of the resultant value when a specific operator is used to evaluate specific operand types.

## ARITHMETIC OPERATORS:

- `*` Multiplication of operands.
- `+` Addition of operands.
- `-` Subtraction of second operand from first.

**Operand** Type: Integer, Long_Integer, or Real.

**Result** Type: If both operands are Short_Integer the result will be Short_Integer.

If one is a Long_Integer and the other is an Short_Integer or Long_Integer, the result will be a Long_Integer.

If one or both operands are Real the result will be Real.

/ Real Division of first operand by second.

**Operand** Type: Integer, Long_Integer, or Real.

**Result** Type: Always Real.

---

**DIV** Integer Division of first operand by second.

**MOD** Modulus of integer division of first operand by second. The modulus is the remainder left over after integer division.

& Bitwise AND of operands.
| Bitwise OR of operands.

**Operand** Type: Short_Integer or Long_Integer.

**Result** Type: Short_Integer if both operands are Short_Integer; otherwise Long_Integer.

---

~ One's Complement of a single operand.

**Operand** Type: Short_Integer or Long_Integer.

**Result** Type: Same as that of operand.

## BOOLEAN OPERATORS:

| | |
|---|---|
| **AND** | Logical AND (conjunction) of operands. |
| **OR** | Logical OR (disjunction) of operands. |
| **NOT** | Logical negation of single operand. |

**Operand** Type: Boolean.

**Result** Type: Always Boolean.

---

## RELATIONAL OPERATORS:

| | |
|---|---|
| **=** | Equivalence of operands. |
| **<>** | Non-equivalence of operands. |
| **>** | First operand greater than second. |
| **>=** | First operand greater or equal to second. |
| **<** | First operand less than second. |
| **<=** | First operand less or equal to second. |

**Operand** Type: Any compatible simple types or STRINGs.

**Result** Type: Always Boolean.

## SET OPERATORS:

| | |
|---|---|
| = | Set equivalence. |
| <> | Set non-equivalence. |
| >= | First operand equal to or superset of second. |
| <= | First operand equal to or subset of second. |

**Operand** Type: SET of compatible base types.

**Result** Type: Always Boolean.

---

**IN** Set inclusion.

**Operand** Type: First operand of ordinal type T, second a SET of base type T.

**Result** Type: Always Boolean.

---

| | |
|---|---|
| * | Set intersection. |
| + | Set union. |
| - | Set difference. |

**Operand** Type: SET of compatible base types.

**Result** Type: Same type as operands.

# PASCAL PROGRAM FORMAT

## INTRODUCTION

This section presents a moderately formal definition of the Personal Pascal language. This is where the structure of Pascal is discussed; we even go so far as to use *syntax diagrams* to show you this structure.

This section will let you determine whether Personal Pascal supports a particular feature or not, but not every part of the structure of Pascal is discussed here. For example, the standard predefined subprograms that are part of *any* Pascal, such as *ReadLn* and *WriteLn* are purposely omitted because a formal presentation of such subprograms would take up far too much space and not be as readable as the informal approach used here. Generally, Personal Pascal implements the standard subprograms as shown in most Pascal tutorial and reference books.

Before we begin looking at the structure of Pascal we need to introduce two topics:

**BLOCK STRUCTURE** page 6-30

**SCOPE** page 6-31

## BLOCK STRUCTURE

Pascal is a block-structured language. This simply means that a Pascal program consists of a single block that may contain within it subprogram blocks, each of which may contain other subprogram blocks, to any number of levels. Let us begin by looking at a trivial but proper Pascal program:

```
PROGRAM Lazy ;

  PROCEDURE Do_Nothing ;
    BEGIN
      ;
    END ;

  PROCEDURE Do_More ;
    BEGIN
      Do_Nothing ;
    END ;

BEGIN
  Do_More ;
END.
```

Do you see the similarities in the three blocks that make up this program? Each has a header that serves to give it a name; Lazy, Do_Nothing, and Do_More, and a BLOCK. In our simple example, each block shown begins with the word *BEGIN* and ends with the word *END.* In a more typical program, each block might also include one or more *DECLARATIONS* but the BEGIN and END keywords would always be there. The only trick to this is that declarations of an *outer* block always precede the definition of any *inner* blocks.

Another way to look at it, as we do in this manual: The definition, or *naming* of an *inner* block is simply part of the declarations of the enclosing *outer* block. Since all declarations must proceed in a certain order, it's reasonable that inner block declarations are the last declarations of a block, before the BEGIN...END portion. Perhaps another topic and example will help...

## SCOPE

SCOPE is shorthand for "What can be referenced or used where." The *What* in that expansion is *DECLARATIONS* of all kinds. The *where* is what we want to explore.

We have mentioned declarations casually, and we will not explore them in detail for several pages, but for now let us consider some *VARIABLES*, as declared in the example on the next page:

## PROGRAM FORMAT ( Continued )

```
PROGRAM Prog ;
VAR A,B,C : Real ;

  PROCEDURE Proc1 ;
  VAR A,D,E : Long_Integer ;

    PROCEDURE Proc1A
    VAR B,D,F : Short_Integer ;
    BEGIN
      ShowTypes
    END ;

    PROCEDURE Proc1B
    VAR A,D,F : String ;
    BEGIN
      ShowTypes
    END ;

  BEGIN
    ShowTypes
  END ;

  PROCEDURE Proc2 ;
  VAR A,B,E : Boolean ;
  BEGIN
    ShowTypes
  END ;

BEGIN
  ShowTypes
END ;
```

In the program on the previous page, we have invented an imaginary predefined subprogram called *ShowTypes* that will, when called:

Show the name of the program or subprogram block that called it.

Show the *TYPES* of all the variables *A, B, C, D, E,* and *F* as they exist in that block.

Call all inner subprograms in turn, to ensure that each of them will have an opportunity to use *ShowTypes.*

Unfortunately, *ShowTypes* is indeed imaginary. In fact, we know of no way within the structure of Pascal to write or invent such a subprogram. But let's pretend anyway. If *ShowTypes* worked properly, this is what we would see:

| Name | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| Prog | Real | Real | Real | ** | ** | ** |
| Proc1 | Long | Real | Real | Long | Long | ** |
| Proc1A | Long | Shrt | Real | Shrt | Long | Shrt |
| Proc1B | Strg | Real | Real | Strg | Long | Strg |
| Proc2 | Bool | Bool | Real | ** | Bool | ** |

Study the example output table closely. Note that ** indicates an undefined and unavailable variable. Do you see what has happened?

In each case the subprogram has inherited the variables of the block that encloses it, excepting that variables defined within that same subprogram have priority. The clearest case of this is variable *A,* which is defined differently in each subprogram except for *PROC1A* where it inherits the *long* type from *PROC1.*

Consider variable *C,* which no subprogram defines and thus is accessible to all levels. Finally, notice that no variable is inherited *outward,* from an inner block to an enclosing block.

It's not just the *TYPES* of variables that are inherited in this fashion. *All* declarations *previously* made by an outer block are available to any inner block. The exceptions: Only those identifiers of the same kind and name that are re-declared in the inner block.

Notice the word *previously* in the above. In our example, *Proc1A* is available to *Proc1B*, but NOT vice versa. Pascal does not normally allow *FORWARD REFERENCES*, but there are some exceptions to and ways around this restriction.

**NOTE:** By convention, declarations made in the main program block, as opposed to within subprogram blocks, are considered *global* declarations. Unless redeclared, their scope is global throughout the program, hence the name.

**ALSO:** Declarations made in subprogram blocks are said to be *local* declarations. However, phrases such as *"local to the declaring block"* apply equally to subprogram and program blocks, since *global* is considered simply a special case of *local.*

## SYNTAX DIAGRAMS

Once we have the elements of Pascal defined, we need to explore its syntax. *Syntax* refers to the well-defined set of rules that comprise a computer language. Any language, including English, has rules of syntax:

understand example for proper without this syntax.

Could you unscramble that to make an understandable English sentence? If *you* have trouble, think how hard it is for a poor Pascal compiler, with its limited brain power. No wonder we need a set of rules! But what good are rules if you don't have a way to describe them?

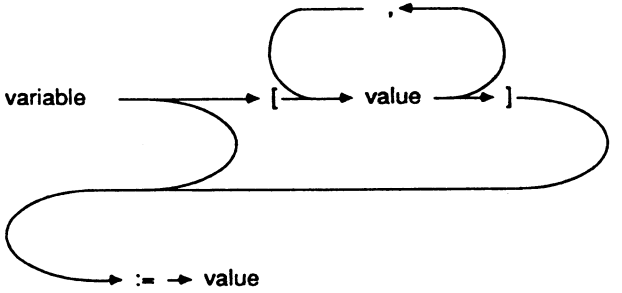
It's possible to describe the syntax of Pascal entirely in English, but it would take an enormous amount of paper, and the result would be an unreadable mess. We need a readable symbolic description.

One of the problems in describing a computer language with symbols is that we tend to want to use the same symbols in the description that the language itself uses in its syntax. For example, here is part of the syntax of Pascal's assignment statement as it would be described in Backus-Naur form, probably the most commonly used symbolic form:

```
assignment ::==
  variable [ '[' value [,value...] ']' ] ':=' value
```

Is that clear to you? Which of those square brackets should you type? What do the other square brackets mean?

Because of the confusion, we have chosen to use language *Syntax Diagrams.* The syntax diagrams used here are often called *Railroad Tracks.* With a little imagination, you can see why. Here is the same partial description of the assignment in railroad track form:



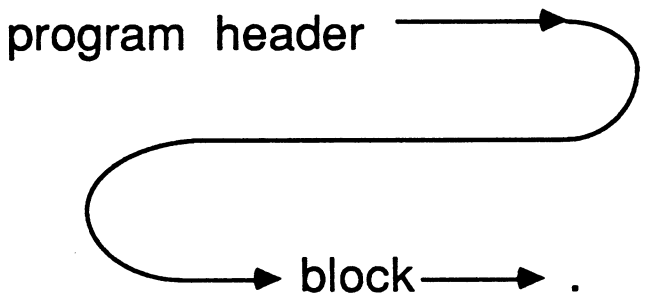

Imagine you are an engine that can only go forward. Start at the beginning of the diagram at the word VARIABLE. Every time you see a branch, you can take it or not. Every time you encounter a symbol or word that is not part of the tracks, except for the arrows which are part of the tracks, you collect that item. All symbols and upper case words are collected exactly as they are shown in the diagram. Lower case words are further explained elsewhere, such as the section on DEFINITIONS.

Put your engine on the tracks and see what valid assignment statements you can make. Among many others, we created these:

```
variable := value
variable [ value ] := value
variable [ value , value ] := value
```

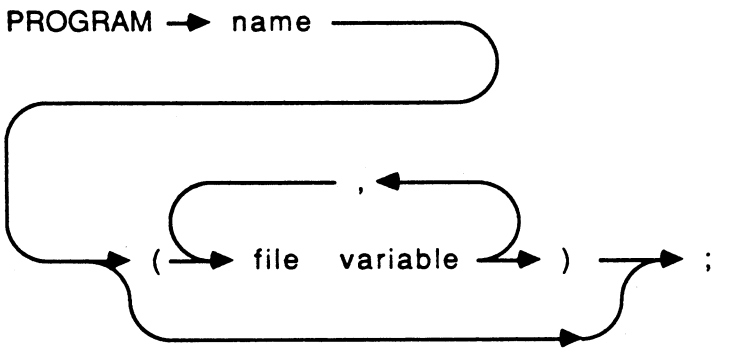
If this is not clear immediately, try reading some of the following syntax diagrams. We tend to keep them simple by using a lot of lower case words that are explained later with their own syntax diagrams.

# PASCAL PROGRAM STRUCTURE




In the discussions of block structure and scope, we have already shown some examples of program structure. Specifically, each program consisted of the word *Program* followed by a name for the program and then followed by a *BLOCK*. The word *Program* introduces a program's *HEADER*, and we need to discuss that further before proceeding.

## PROGRAM HEADER

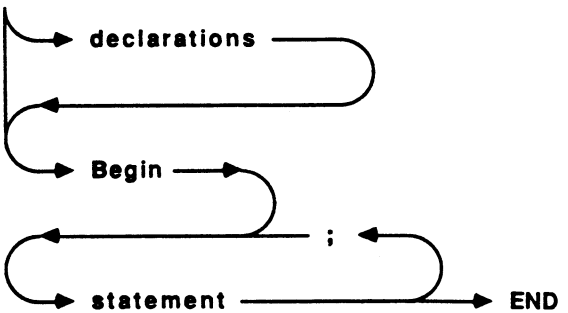



As the diagram reveals, there can be more to a program header than the word *Program* followed by a name. Specifically, we might enclose one or more file variables in parentheses before the semicolon that terminates the header.

Standard ISO Pascal provides for PROGRAM PARAMETERS. For compatibility, Personal Pascal allows these parameters, but *completely ignores* them, since there is no standardized way in the TOS/GEM environments to associate particular files with the given file variables.

**CAUTION:** If any file variables are given in the header, they must be properly declared global variables. The exceptions: In accordance with ISO Pascal, the file variables *Input* and *Output* are always available as pre-declared as *Text* variables.

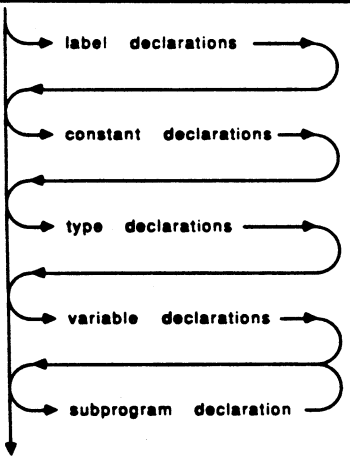
## PASCAL BLOCKS, PART I




We have already noted the general structure of Pascal blocks: A block consists of optional declarations followed by a BEGIN...END pair that enclose one or more statements. The enclosed statements are *separated* by semicolons not *terminated* by them, as we shall see later.

Until we look at how declarations are made, there is only one comment to be made here: Note in the Pascal Program Structure diagram, that the block of any main program is terminated by a period. As we will see later, subprogram blocks are terminated by a semicolon. Aside from the *form* of subprogram headers, this is the sole discernible difference in Pascal's neatly recursive block structure.

# DECLARATIONS

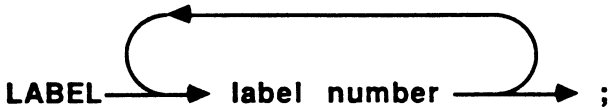



Within any given standard Pascal block, as many as five different kinds of things may be declared. Label, constant, type, and variable declarations are all grouped, in that order, under an appropriate heading keyword. Subprogram declarations are grouped last, but each subprogram has its own header to distinguish it.

**SPECIAL FEATURE:** Personal Pascal version 2 allows these declarations in any order, with repetitions of any kind of declaration allowed, so long as each repetition is preceded by the appropriate keyword. This makes *Include files* easier to work with. Note that, in general, *forward references* still are not allowed.

Remember the scope of declarations: All inner blocks have access to all declarations of the enclosing block, unless they make a redeclaration of the same name and kind.

## LABEL DECLARATIONS

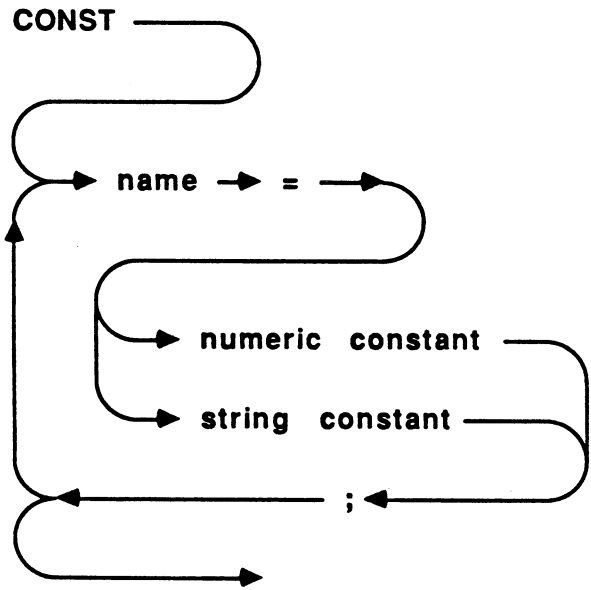



The declaration of labels is extraordinarily simple. In each block, labels local to that block are declared one after another, with separating semicolons, following the keyword *LABEL.*

**EXAMPLE:**

```
PROCEDURE Proc1 ;
LABEL
   0 ; (* emergency exit *)
 100 ; (* user typing error *)
```

## CONSTANT DECLARATIONS




If there are values that do not change throughout the duration of your program, you might give them names rather than having them crop up in your program as magic numbers. Constants given names as described here are called DECLARED CONSTANTS, or sometimes PROGRAM CONSTANTS. Using declared constants offers several advantages, including:

Increased portability
Documentation of implementation-defined values
Setting program-specific bounds and limits.

Personal Pascal supports two types of declared constants: Numbers and Character Strings. Declared constants can be used in several places:

When defining program constants using CONST
When assigning values to variables
When comparing current variable values to expected ones

In short, wherever an ordinary constant of the same type can be used.

This is an example of a valid set of constant declarations:

```
CONST Minint = -Maxint ;
    pi = 3.1415926535 ;
    pie = 'pie' ;
    our_wives = 'Bev and Barb' ;
    mole = 6.022E23 ;
```

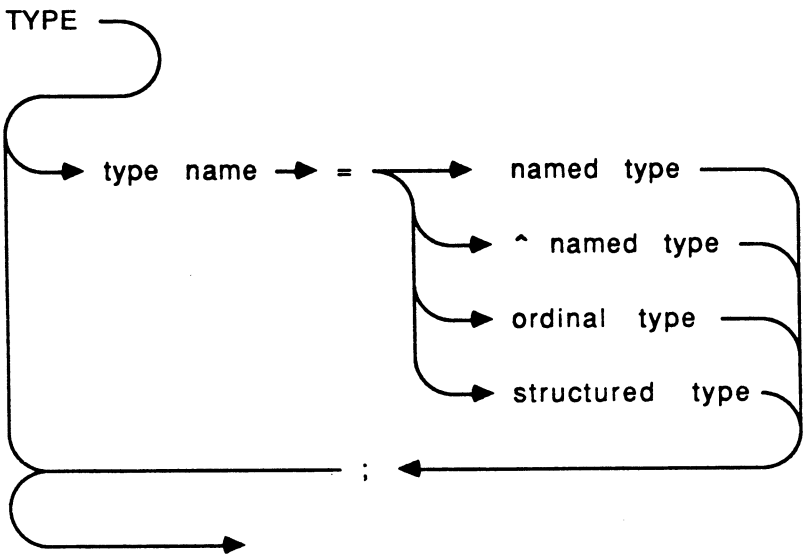
A declared constant may not reference itself in its definition ; for example:

```
CONST Bad = -Bad ;
```

is illegal.

**NOTE:** The symbols '+' and '-' may only be applied to constant identifiers that refer to numeric constants ; not character strings.

# TYPE DECLARATIONS




Sooner or later you will want to create your own data types ; either mixtures or ranges of predefined types, or a new type altogether. You can do this in the type definition section of a block. The general form of the type definition is shown in the railroad tracks above, but we will break it down further. Before doing so, we need to look at a topic referred to in several places in this section, even though its importance won't be obvious until later sections where assignments are discussed.

## TYPE COMPATIBILITY

Pascal imposes compatibility rules on data types to insure that data manipulation remains consistent within a program. This compatibility becomes important on two occasions: when passing parameters or when assigning values to variables. Two variables are type compatible if any one of the following is true:

They are of the same type.

Both are subranges of the same ordinal type

One is a subrange of the other

Both are *SETS* or *PACKED SETS* and have the same ordinal base type.

They are *STRINGS* of the same declared size.

The additional rules regarding assignment are called Assignment Compatibility.
**SEE:** *Assignment Compatibility,* page 6-105

## NAMED TYPES

Named types are data types that have a name, either predefined by Personal Pascal or defined in a preceding TYPE section in your program. When you define two variables as the same type by name, you ensure type compatibility between them.

As shown in the syntax diagram on page 46, you may define a type as being equal to a named type name. That is, a type identifier may be made to become equivalent to another type name.

The *predefined* Personal Pascal types which may be used whenever a named type is needed are

| | | |
|---|---|---|
| **Alfa** | **Byte** | **Boolean** |
| **Char** | **Integer** | **Short_Integer** |
| **Long_Integer** | **Real** | **String** |
| **Text** | | |

**SEE:** Examples on the next page

Some examples of using named data types in a type declaration are:

```
TYPE
   payment = Real ;
   total   = payment ;
   name = String ;
   count = Short_Integer ;
   subcount = count ;
```

**SPECIAL NOTE:** Standard Pascal does not define or support Personal Pascal's *Short_Integer* or *Long_Integer* data types. Instead, only the generic *Integer* type is defined. By default, Personal Pascal treats all references to Integer as if the you had coded *Short_Integer*. However, you can use the {$I+} compiler directive to force references to Integer to become equivalent to *Long_Integer.* This flexibility allows you to produce programs which are as compatible as possible with various other Pascal compilers. Naturally, variables of type *Short_Integer* cause faster program execution and use less memory than those of type *Long_Integer.*

## POINTERS TO NAMED TYPES

A type identifier may also be used to denote a pointer to an existing named type. Declaring such *POINTER TYPES* is easy. Some examples of declaring pointer types are:

```
TYPE
    Char_Pointer = ^CHAR ;
    Node_Pointer = ^Node ;
    Node = RECORD
      value : STRING ;
      leftchild, rightchild : Node_Pointer ;
      END ;
```

Notice that the Node_Pointer's domain-type, what it points to, is defined after the pointer type itself. This is legal only if the domain definition is in the same TYPE section as the pointer section. The advantage is that it's easy to create linked structures, such as trees and lists.

## NEW TYPES

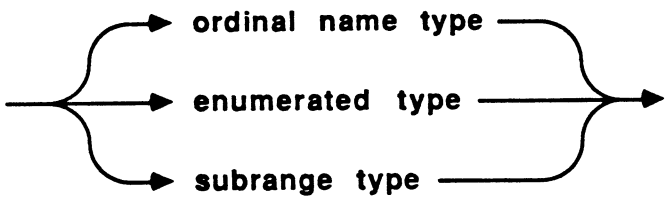
Though not shown in the syntax diagram above, the term NEW TYPES is used throughout this manual, so an explanation is in order.

A *new* type is anything that isn't a *named* type. If you declare a variable using a type that consists of anything other than the name of an already defined type, you are specifying a new variable type.

**CAUTION:** In standard Pascal, new structured types are generally *not* compatible with other *named* types, even if they are declared identically. Personal Pascal is a little more liberal in its type compatibility and assignment compatibility rules. Some permissible Personal Pascal record assignments, for example, will not be portable to other Pascal compilers.
**SEE:** *Referencing Records*, Page 6-82

**NOTE:** Your programming style will become more elegant, and readable if you define all of your types in the *TYPE* section and then use only named types when declaring variables. One of Pascal's virtues is the outstanding readability of properly written programs.

## ORDINAL TYPES




As the syntax diagram shows, there are three kinds of ordinal types. The first kind is simply the name of an already-defined ordinal type. There are six predefined ordinal types in Personal Pascal:

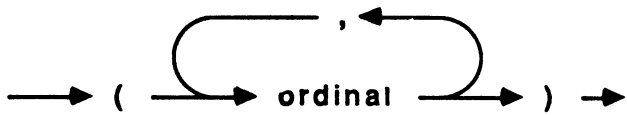
| | | |
|---|---|---|
| **Byte** | **Boolean** | **Char** |
| **Integer** | **Short_Integer** | **Long_Integer** |

**SEE:** *Definitions*, page 6-12 for the ranges of values these types may take.

In addition, any ordinal types that you create, in accordance with the rules of this section, become named ordinal types.

The other two *new* ordinal types are always programmer declared: *enumerated* types and *subrange* types.

## ENUMERATED TYPES




An enumerated type is simply a data type whose elements are each defined with an identifier.

**EXAMPLE:**

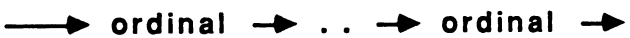
```
TYPE Rainbow = (Red, Orange, Yellow, Green, Blue,
                Indigo, Violet) ;

VAR filter : Rainbow ;

filter := Blue ;
```

The ordinal, ( *Ord* ), predecessor ( *Pred* ), and successor ( *Succ* ) values for an enumerated type depend entirely upon the order in which the elements are listed in the type definition. In our example,:

```
Ord( Red ) is 0
Pred( Blue ) is Green
Succ( Indigo ) is Violet
Ord( Violet ) is 6
```

Note that the *Ord* value of the *first* element in the list is *0*, not *1.*

## SUBRANGE TYPES




Pascal offers an extension to ordinal types that can also increase the readability of your programs. This extension is subranges of ordinal types. If you were manipulating only upper case letters, you could declare a variable like:

```
VAR achar : CHAR ;
```

This does not imply that you are using only upper case letters. Using a subrange of the *Char* type can make this obvious:

```
TYPE Caps = 'A'..'Z' ;

VAR achar : Caps ;
```

This form directly implies that you're using only upper case letters.

Both of the constants must belong to the same ordinal data type, and the first constant, the low boundary, must be less than or equal to the high boundary of the subrange. Also note that the constants may be of an enumerated ordinal type, as shown in this example:

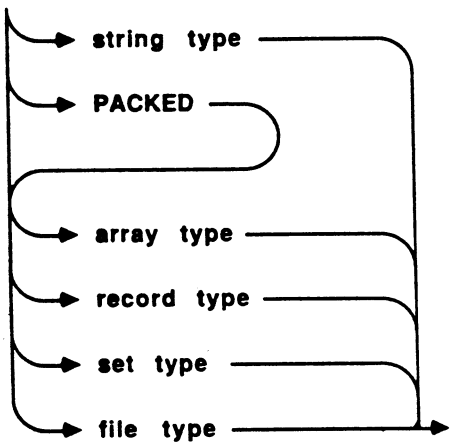
```
TYPE Reds = Red..Yellow ; { type Color }
```

**SPECIAL NOTE:** We stated that Byte was a predefined ordinal type. Actually, Byte is traditionally described as a predefined subrange, as though you had made a declaration such as this:

```
TYPE Byte = 0..255 ;
```

*Byte* simply represents *positive Short_Integer* values that can be contained in one byte. In *PACKED ARRAYS* or *RECORDS*, restricted subranges, such as Byte, may use less memory.

# STRUCTURED TYPES




The simple data types described above cover information that Pascal considers as single units, but we frequently think of more complex information as a single piece of data. A good example is a checking account entry. Although it's one entry, it's made up of several pieces of information, including:

**DATE OF TRANSACTION**
**TRANSACTION NUMBER**
**TRANSACTION TYPE**
**CHECK NUMBER**
**PAYEE**
**PURCHASE MEMO**

The structured types in Pascal allow you to group items like this together so that your program can more closely model human reality.

Personal Pascal supports five different structured forms of data.

**STRING**s are similar to arrays, but are designed especially for character data. In addition, Personal Pascal provides several built-in subprograms and operators that let you manipulate string data easily.
**SEE:** page 6-59

The **RECORD** type allows you to group information of different data types, as in our checking account example. Records are very useful when describing complex information with many attributes.
**SEE:** page 6-61

The **ARRAY** groups information of the same data type together, and is useful when creating lists of similar data, like test scores.
**SEE:** page 6-68

The **FILE** type gives you access to external devices like disk drives and printers, so you can create and read data that exists outside your program.
**SEE:** page 6-71

The **SET** type lets you collect data into a group which can then be compared to other groups. Unlike arrays and strings, sets have no inherent order. A SET has no first element or last element.
**SEE:** page 6-74

All of the structured types but STRING can be PACKED. Packing a structured type generally reduces the amount of space occupied by the type, but often increases the time required to access items within the structure. If you declare a type as PACKED, you should keep these rules in mind:

A *PACKED ARRAY OF CHAR* is not the same as *STRING.*

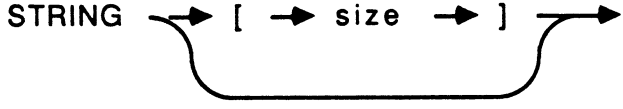
Two structures that are the same except that one is *PACKED* and the other is not, are *NOT* type compatible, although their components are.

Components of *Packed* variables may not be used as actual parameters to VARiable formal parameters.

The procedures *PACK* and *UNPACK* may only be used with *ARRAY* type data.

Because the structured types are declared and accessed differently, each is described in its own section.

## STRING TYPE




Personal Pascal provides a structured data type designed to manipulate text: the *STRING* type. It's similar to *PACKED ARRAY OF CHAR* but has several important differences:

A *STRING* type defaults to a maximum of 80 characters unless its size is declared.

Strings have a length associated with them that may be equal to or less than a string's maximum size.

There are several built-in subprograms that perform text oriented operations on String variables.

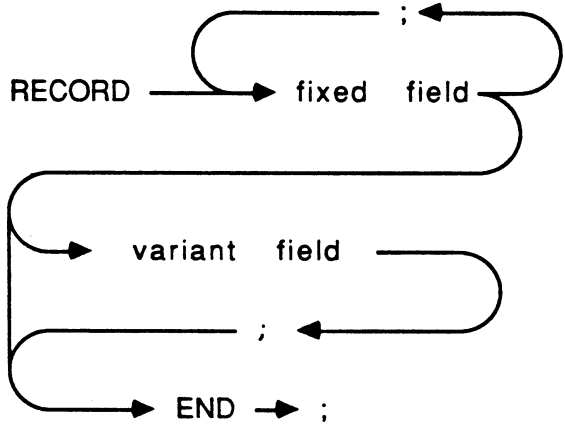
The comparison operators work on Strings in a predictable and orderly fashion.

To declare a string's maximum size, simply place that size within square brackets after the word 'STRING' in the type declaration. For example:

```
TYPE
   footnote = STRING [ 40 ] ;
   names = STRING [ 25 ] ;
   biggest_string = STRING [ 255 ] ;
   standard_string = STRING ;  { 80 is default }
```

**NOTE:** Strings may not be declared to be longer than 255 characters. There is no such limitation on CHAR arrays, but the string manipulation routines and operators do not work with CHAR arrays.

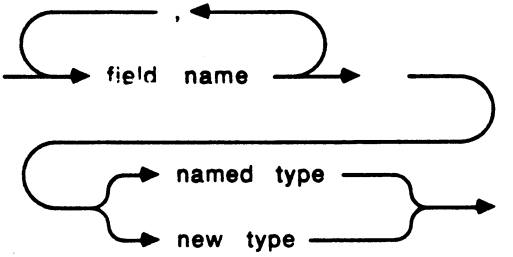
## RECORD TYPE




A RECORD is a structure made up of fields, each having its own name and data type. These component fields may then be referenced using the name of the record variable and the name of the field.

There are two types of field lists - fixed and variant. Fixed records are simpler, so let's discuss them first.

## FIXED FIELDS



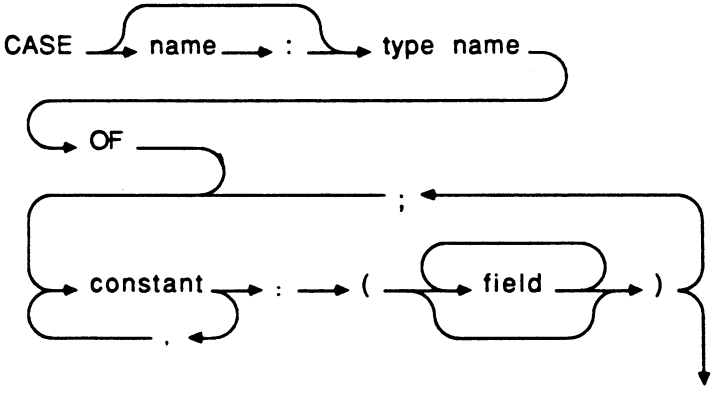

Records which contain only fixed fields are called *FIXED RECORDS*. Such records are completely defined and may not change form while the program is running. Some examples of fixed RECORD types:

```
TYPE TransType = (Withdrawal,Deposit,Transfer) ;

     DateType = RECORD
                mon : 1..12 ;
                day : 1..31 ;
                yr : 0..99 ;
                END ;

     Transaction = RECORD
                   date : DateType ;
                   trunum,
                   trID : integer ;
                   trType : TransType ;
                   amount : Real ;
                   END ;
```

The identifiers within a RECORD definition have a very small scope - between the RECORD and the END - so the following example does NOT create an error despite the duplicate use of the identifier *base*:

```
TYPE Rectangle = RECORD
                 base,
                 height : Real ;
                 END ;

     square = Rectangle ;
     base = Short_Integer ;
```

## VARIANT FIELDS




Fixed records can become cumbersome in certain situations. Our Transaction record, in the fixed record example, would be more useful if we could specify information based on what kind of transaction is being done. Variant records, containing variant fields, let you do that easily.

The method used to define a variant field looks a little like a *CASE* statement, except that the *ELSE* and *OTHERWISE* options are not legal.

```
TYPE   member_type = ( current, expired,
non-member ) ;

DateType = RECORD
        mon : 1..12 ;
        day : 1..31 ;
        yr : 0..99 ;
        END ;

phone_list = RECORD     { Record begins here. }
         name : String ;          { This part is fixed. }
         age : Short_Integer ;
         CASE membership : member_type OF
                        { Variant field here. }
            current : ( renew_date : DateType ;
                    rank : String ;
                    dues : Short_Integer ) ;
            expired : ( exp_date,
                    last_contact : DateType ) ;
            non-member : ( guest_of : String ;
                    last_contact : DateType ) ;

          END ;                { Record ends here. }
```

Notice that the phone_list record has both fixed and variant parts. What makes this record vary is the tag, *membership.* When you change the value of tag field while the program is running, the variant corresponding to the value of tag field becomes active, and the record contains the field list specified by that variant.

As you might suspect, a variant field must be active before it's accessible. This means that the tag field's value must coincide with the proper variant first.

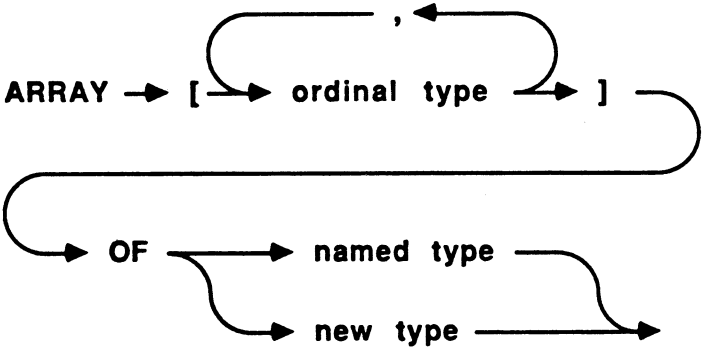
**NOTE:** The tag field is optional ; only a tag type is required. Omitting a named tag field creates a variant part in which all variants are accessible. In essence, the variants overlay each other. This is useless in most cases, but one important use of it is to change pointer data types, which is otherwise impossible. The following function *peeks* at a word-sized Short_Integer memory location:

```
TYPE    IntPtr = ^Integer ; { for convenience }
        Either = 0..1 ; { small enumerated type }
        TwoWay = RECORD
                 CASE Either OF
                 0 : ( where : Long_Integer ) ;
                 1 : ( iptr  : IntPtr ) ;
                 END ;

FUNCTION  IPeek ( location : Long_Integer ) : IntPtr ;
   VAR coerce : TwoWay ; { a funny record! }

   BEGIN
     coerce.where := location & $FFFFFE ; { to ensure
word address }
     IPeek := coerce.iptr^ ;
   END ;
```

Because the variant part of *TwoWay* is controlled by a tag type but not a tag identifier, both variants are always active and may be used at any time.

**CAUTION:** Unless you ask the compiler to ignore pointer checking, this example will only work if the peeked location is within your program's heap space.
**SEE:** *Compiler Directives*, page 4-9

## ARRAY TYPE




The array is a structured type, like the record, but its components have several restrictions. Unlike record fields:

Array components are unnamed.
Array components must all be of the same type.
Array components have an innate order.

Arrays are useful structures because of these restrictions.

Here are some examples of defining ARRAY types:

```
TYPE
   OneByFour = ARRAY [1..4] OF Short_Integer ;
   FourByFour = ARRAY [1..4] OF OneByFour ;
   Friends = PACKED ARRAY [1..100]
             OF STRING [20] ;
```

Frequently you see array definitions of the form:

```
ARRAY [1..100]
```

This index-type is a subrange, and does not merely specify the bounds of the array ; it also says that the array components will be accessed using Short_Integer subscripts.

Notice that more than one index is allowed. An array that has n indices is called n-dimensional. FourByFour, above, is an example of an array of arrays. Another example:

```
TYPE
  TicTacToe_3D = ARRAY [1..4,1..4,1..4] OF BYTE ;
```

Note that the syntax diagram shows that the index-type may be any ordinal type. In theory, this is true. In practice, the ST computers do not have enough memory to handle a declaration of forms such as this example:

```
TYPE
  Monster = PACKED ARRAY [ Long_Integer ] OF
            CHAR ;
```

This particular example would require over four billion bytes of memory for an array of type Monster. In practice, index-types are virtually always either subrange or enumerated types.

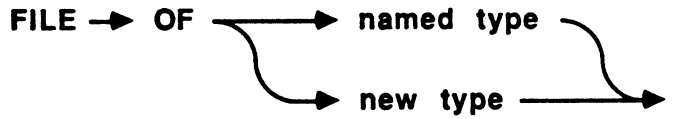
## ALFA TYPE

Personal Pascal provides one *predefined ARRAY* type: *Alfa.* This type is defined as:

```
TYPE Alfa = PACKED ARRAY [1..10] OF CHAR ;
```

Please do not confuse this type with *STRING*; they are not they same. Alfa is of limited use and is provided only to aid compatibility with other less complete versions of Pascal.

## FILE Type




Of all the data types available in Pascal, FILE may be the most important because it is the only type whose data can exist before a program runs, or after it's finished. This is because FILE type is used to interface between a Pascal program and external storage devices such as disk drives and printers. Unlike other variables, a FILE variable doesn't have a size associated with it because the external device is the only limiting factor to a FILE's size. Declaring a FILE type is easy:

```
TYPE
  payroll = RECORD
        name : string[25] ;
        date : Long_Integer ;
        amount: Real ;
  END ;
  tally : FILE OF Integer,
  lines : FILE OF String [255] ;
  payfile : FILE OF payroll ;
```

**NOTE:** The component type of a FILE may not be or contain a FILE type.

Declaring a variable of a *FILE* type creates a buffer variable of the file's component type, but doesn't immediately make that variable accessible. Before you can access the buffer variable, the FILE variable must be associated with an external device. You do that using the Reset or Rewrite procedures, depending upon whether the file is to be used for input or output.

The buffer variable exists as temporary storage for a single component of a file. It increases program speed because accesses to pieces of a structured file component are done more quickly when the component is in a buffer rather than on an external device.

The only way to transfer data to or from the physical file is through the buffer variable. As a result, the data transfer subprograms for FILE type variables always manipulate the file buffer, either directly or indirectly.

**NOTE:** Throughout this manual we use the term *FILE IDENTIFIER* to collectively indicate the file variable and so-called buffer variable, neither of which really resemble conventional variables.

## SPECIAL FILE TYPE TEXT

Text is a type designed specifically for text files. A Text file is very similar to a PACKED FILE OF Char. Personal Pascal has special subprograms that manipulate Text files on a line-of-text basis.

Note: In Personal Pascal, the end-of-line character is internally represented by the ASCII CR ( $0D ) and LF ( $0A ) characters. However, when you Read or Get the end-of-line from a TEXT file, it's translated into a space. The end-of-page character is internally represented by the ASCII FF ( $0C ) character, and is also translated into a space on READ or GET.

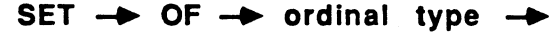
Generally, peripherals other than disk files should be declared as FILEs of type TEXT. For example:

```
TYPE
  printer_type = TEXT ;
```

Notice that you do NOT use "File of Text" since TEXT is already a file type.

**NOTE:** The predefined file variables *Input* and *Output* are assumed to be of type *TEXT.*

## SET TYPE




The SET type provides a structure not available in most programming languages: a grouping of ordinal data that has no inherent order. These are examples of SET definitions:

```
TYPE Senior = 65..100 ;
     Caps = SET OF 'A'..'Z' ;
     Retired = SET OF Senior ;
     AllChars = SET OF CHAR ;
```

**NOTE:** In Personal Pascal a set may have up to 128 members. For purposes of sets, type *CHAR* is considered to have 128 values ( #0..#127 ). For most other purposes, *CHAR* has 256 values ( #0..#255 ).

# VARIABLE DECLARATIONS

## VARIABLES AND DATA TYPING

Variables are places in memory where values can be stored. Pascal places these restrictions on the use of variables:

A variable is associated with a particular data type and can only be used to store values of that type.

A variable must be declared by name and data type in a variable declaration or formal parameter list before you use it.

The scope of a variable is restricted to the block that contains it, and its lifetime is limited to the time that the block containing it is actually running.

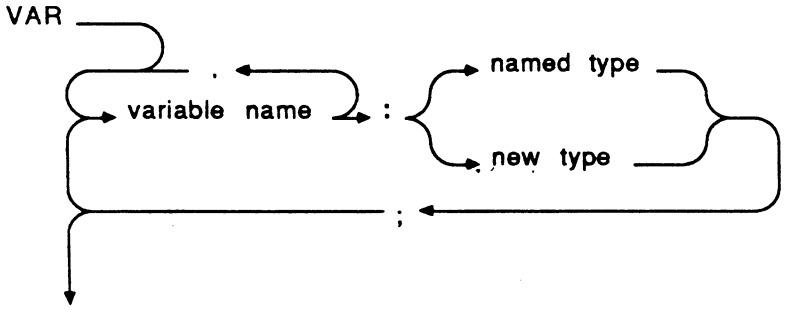
**NOTE:** Variables declared in the main program block also follow this rule: their lifetime is the life of the running program.

You must declare the data type of every variable that you use in a Pascal program before you begin to use it. The variable declaration section is where you do this. A simple example is:

```
VAR
  income  : Integer ;  { Integer is the TYPE }
  expense : Long_Integer ;
```

Types can be either a named type or a new type, as described in the section on TYPES.

## VARIABLE DECLARATIONS




At the beginning of a program no variables exist. Global variables, declared in the program block, are created first. Memory space is reserved for them, but they are given no initial value ; they are undefined. Structured variables are totally undefined because none of their components are defined. They become defined when you assign values to them. The lifetime of global variables is the entire time the program is running, whether they are ever defined or not, as noted above.

When the program calls a procedure or function, the variables declared in the subprogram are created, and they exist until the subprogram terminates and returns control to the main program. If one subprogram calls another, the variables in the called subprogram are created, and exist until control returns to the calling subprogram.

## HOW VARIABLES WORK

The general form used to declare variables is shown above. Here are examples of variable declarations:

```
TYPE
  Age_Range : 0..99 ;
VAR
  rbi : Integer ;       { named type }
  avg,income : Real ;   { named type }
  games : 0..152 ;      { new type }
  age : Age_Range ;     { named type }
  name : STRING [20] ;  { new type }
  Oops : ARRAY [ 1..40 ] OF Integer ;  { new }
```

In this example, Integer, Real, and Age_Range are *NAMED* types, and the other three are *NEW* types. Remember the rules for *TYPE* compatibility: the array *Oops* may not be assigned to or from any other array, even one which is declared identically, because they can not have the same *TYPE NAME*. *ELEMENTS* of the *Oops* array may be used any place an integer value or variable is required.

## KINDS OF VARIABLES

Pascal offers several ways to access a variable. Variables can have one name, several names, or none at all. The following situations create these different kinds of variables:

> Variable declarations and value parameter declarations allocate new variables with a single name.
>
> Variable parameter declarations create a synonym for an already existing variable.
> **SEE:** Procedure and Function Declarations.
>
> The *NEW* subprogram allocates variables that have no name. *FILE* type variables contain an unnamed buffer variable.

A variable can be made up of several components, as in the case of a *RECORD* or *ARRAY*. In that case, it is a structured variable and the components are accessible only as part of the parent, not as separate variables.

When a variable has a name, as in the first 2 cases above, its identifier is called the *entire variable* because it refers to the whole variable, whether the variable is simple, having a single component, or structured. Using its name is only one of the ways to access a variable.

## VARIABLE DECLARATIONS ( Continued )

Here are all the ways to access variables:

### THE ENTIRE VARIABLE:

```
e.g.: array_name
      total_time
```

### BY COMPONENT:

```
e.g.: answer [ 4, True ]
      { a 2-dimensional array }

      Matrix [ xdim, ydim, zdim ]
      { a 3-dimensional array }

       check.number
      { the number field of a check record }
```

### AS BUFFER VARIABLE FILE POINTERS:

```
e.g.: datafile^          { a file pointer }
```

### AS DYNAMIC VARIABLES:

Un-named variables created by the *New* subprogram.

```
e.g.: NEW( charpointer ) ; { followed by ... }
      charpointer^   { a pointer }
```

**NOTE:** The various methods of variable access are recursive. This can result in variable accesses that are almost incomprehensible:

```
x := employees[ y ].personal.children[ z ]^.name
```

This variable access can be explained in English:

*Employees* is an array of a record type that has a field called *personal. Personal* is a record type that has a field called *children. Children* is an array of pointers to a record type that has a field called *name.*

Why such a complex variable access? Well, consider the complexity of employee records:

We have several employees: - *employees[y]*

We have business and personal information about each employee: - *.personal*

Some employees have children: - *.children[z]*

We keep information about each child in a dynamic record, including the child's name: *^.name.*

## REFERENCING ARRAYS

Arrays can be referenced either whole or by component. If two variables are of the same array type, as in:

```
VAR v1,v2 : ARRAY [1..1000] OF Short_Integer ;
```

you can assign one to the other with a simple assignment::

```
v1 := v2 ;
```

This statement copies the entire array v2 into v1.

When two variables are of different array types, assignment between the two can be made if the component types of the two array types are compatible:

```
VAR v3 : ARRAY [1..10] OF Short_Integer ;
    v4 : ARRAY [1..5,Boolean] OF Short_Integer ;

v3[8] := v4[2,True] ;
v4[1,False] := v3[1] ;
```

When accessing arrays by component, the expressions used to define the indices must be compatible with the data types of indices In the example above, v4[False,1] would be invalid.

## REFERENCING RECORDS

As with arrays, records may be referenced either as a whole or by field. The rules in standard Pascal for referencing whole records are fairly restrictive. Consider this example:

```
TYPE
  Rtype = RECORD
        ri : INTEGER ;
        rc : CHAR ;
        END ;
VAR
  Rec1 : Rtype
  Rec2 : Rtype
  Rec3,Rec4 : RECORD
            ri : INTEGER ;
            rc : CHAR ;
            END ;
```

These assignments are legal:

```
  Rec1 := Rec2 ;
  Rec3 := Rec4 ;
```

These assignments will be flagged by standard Pascal as errors, but are acceptable to Personal Pascal:

```
  Rec1 := Rec4 ;
  Rec3 := Rec2 ;
```

The two record types seem to be completely compatible: they are the same size, with the same number and type of fields, even with fields named the same. Although standard Pascal does not go so far as to analyze the records' contents, Personal Pascal does so. If two records consist of identical root components, Personal Pascal considers them both type and assignment compatible.

In any case, record FIELDS of compatible types may always be assigned. These are always legal:

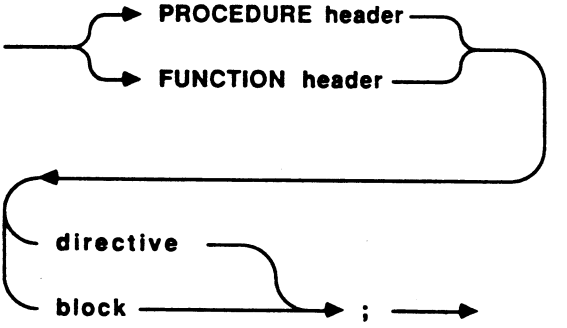
```
Rec1.ri := Rec4.ri ;
Rec3.rc := Rec2.rc ;
```

Remember: The SCOPE of field names is very limited. They are accessible only within the record type where they were declared:

```
VAR
  Rec5,Rec6 : RECORD
              rc : INTEGER ; (* Look closely!)
              ri : CHAR ;    (* Look closely!)
              END
  Rec1 := Rec5 ; { Not legal in standard Pascal! }
  Rec1.ri := Rec6.rc ;
  Rec5.ri := Rec4.rc ;
```

It doesn't look good, and we certainly don't recommend such declarations, but that example is perfectly legal and will work fine. Messy, isn't it?

# SUBPROGRAM DECLARATIONS




When you write a program, you have some task in mind that you wish to accomplish. To perform that task, you break it up into sub-tasks, performed in a certain order, subject to certain conditions. The description of these sub-tasks and their order of performance is called an algorithm.

If you make your Pascal program follow the steps of your algorithm, then the logical thing to do is write subprograms that perform the sub-tasks. Of course, it is logical that sub-tasks are broken down into further sub-tasks, implemented in turn by other subprograms. The algorithm is then accomplished by the program calling subprogram which in turn may call other subprogams, and so on.

Pascal implements two types of subprograms: *PROCEDUREs* and *FUNCTIONs*. The primary difference between the two is that *FUNCTIONs* return a value of some kind to the caller. It may be more correct to say that each function *has* a value, much as a variable has a value, that may change depending on what has gone before. The syntax of Pascal requires that anything with a value *must* be used in an expression. *PROCEDUREs*, on the other hand, may not be used in expressions and have no inherent *value*, so a call to a procedure simply becomes a statement in the block of a program or subprogram.

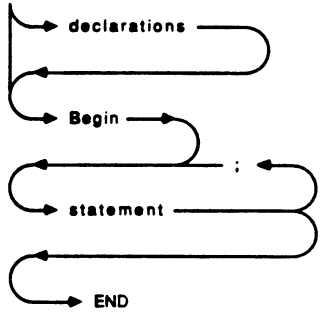
## DIRECTIVES

There are two primary forms for declaring subprograms, as shown in the syntax diagram at the beginning of this section. The first method is the simpler: You simply declare the header for a *PROCEDURE* or *FUNCTION* and then follow it with a language directive.

Since the available language directives *FORWARD, EXTERNAL, C, GEMDOS, BIOS*, and *XBIOS* were discussed in detail in the section on *Definitions*, we won't rehash them here. We will note, again, that of these only the *FORWARD* directive is supported in ISO Pascal.

We also suggest that you examine the *GEMSUBS.PAS* and *AUXSUBS.PAS* files for some examples of these directives in use.

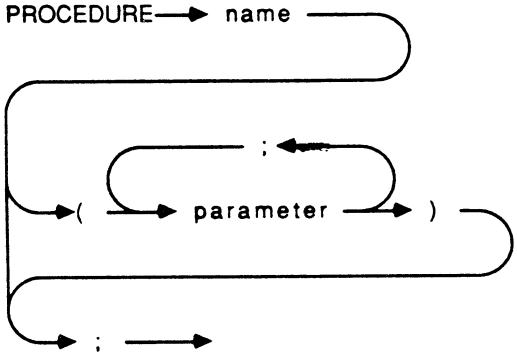
## PASCAL BLOCKS, PART 2




The second and most usual form of subprogram declaration is so much like writing the main program that the syntax diagram shows only the word *BLOCK* after the appropriate header. As we will note on the next page, *subprogram* headers are *very* different from program headers, but the *BLOCK* is identical. Only the terminating semicolon in place of a program's terminating period shows that this block is a subprogram block.

**Remember:** a Pascal block is always recursive. The first thing in the block is *DECLARATIONS* which can, in turn, include subprogram declarations, which can in turn declare still more nested subprogram declarations.

## PROCEDURE HEADER




Procedure headers have a deceptively simple looking syntax diagram. True, for the simplest procedures the header consists of nothing but the word *Procedure* followed by a named identifier, but when a procedure starts using parameters things can get complex quickly.

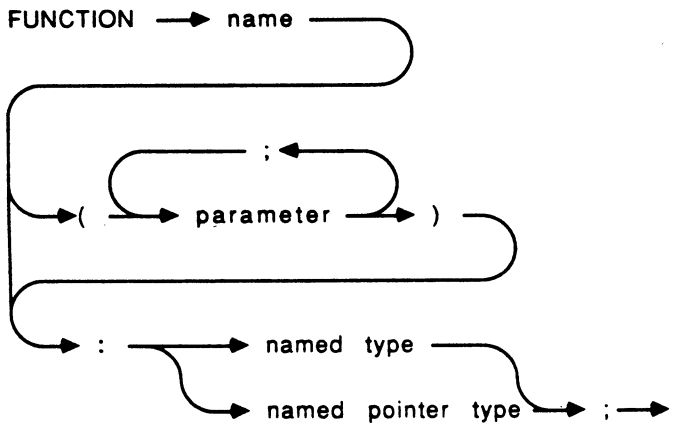
The purpose of parameters is to provide a consistent way for subprograms to receive or process information supplied by a caller, whether the main program or another subprogram. Since procedures, unlike functions, do not automatically return a value to the caller, many procedures simply accept information, process it and quit, returning to the caller. However, a procedure may modify the information given it by the caller if one or more of the parameters is a variable parameter, as described later.

## REFERENCING PROCEDURES

A procedure is referenced by the calling program or subprogram by simply making it a statement somewhere in the caller's block. As a specific example, here is a complete, if somewhat silly, program with a procedure declaration and reference. This procedure does not use parameters.

```
PROGRAM Example ;

(* 'Stars' simply displays 15 asterisks *)
PROCEDURE Stars ;
  BEGIN
  WriteLn( '***************' )
  END ;

BEGIN
  (* first, call predefined subprogram *)
  WriteLn( 'Here are some stars' ) ;
  (* then call our own subprogram *)
  Stars
END .
```

## FUNCTION HEADER




The only real difference between a procedure header and a function header is that a function header ends with a colon followed by a named type. The type name that you declare here, or that is declared for you in the case of predefined functions, describes the type of value that this function will return. Note that only the simpler data types may be returned by a Pascal function. In particular, only *named Ordinal* and *Real* types may be returned, except that a *pointer* to any kind of type may also be returned.

## REFERENCING FUNCTIONS

The calling program or subprogram must DO something with this value, which means that the function must be called within an expression of some kind. For most purposes, this means that a function may be treated as a variable. Here is a simple program that declares and then calls a function subprogram:

```
PROGRAM Demo ;

FUNCTION Square( Number : Integer ; )
         : Integer ;
  BEGIN
    Square := Number * Number
  END ;

BEGIN
  WriteLn( '7 times 7 is ', Square( 7 ) )
END .
```

## SUBPROGRAM PARAMETERS

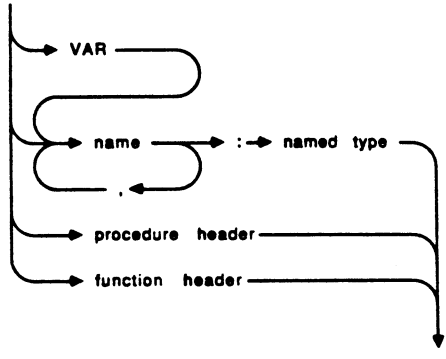
We use the word *parameter* to mean both the value or variable or subprogram designator which a subprogram *receives* from the caller, as well as the value or variable or subprogram name *given* by the caller. In the last example program, the variable *number* is the receiving parameter in the *square* function; the value 7 is the given parameter. Naturally, Pascal has names for these two type of parameters: *ACTUAL* and *FORMAL:*

## ACTUAL PARAMETERS

When you call a procedure or function that requires you to give it one or more values or variables in order to perform its task, the data you pass are called the *ACTUAL* parameters. The number and types of actual parameters that you pass must match the parameter list given in the subprogram's header. In this case, the word *match* means to be *TYPE COMPATIBLE.*

Some types of parameters have special rules, as discussed further on.

## FORMAL PARAMETERS




Formal parameters are the variables that you name and create in the header declaration of a procedure or function. When a subprogram with formal parameters is called, the caller must supply values, variables, pointers or subprogram names, depending on the declaration of a particular formal parameter.

Technically, formal parameters are not variables. But from a coding point of view, excepting for subprogram designators, there is little discernible distinction between the two. In fact, if you look at the syntax diagram for variable declarations and compare it to the top part of the diagram above, the only difference seems to be where and how often the VAR keyword appears. But there is a major difference: In the variable declarations, VAR serves only as an introductory keyword. In formal parameters, the presence or lack of VAR distinguishes two completely different methods of passing parameters.

## VALUE PARAMETERS

When the *VAR* keyword does *not* appear ahead of a parameter's name in a subprogram header, the parameter is said to be a value parameter. During program execution, Pascal allocates space for a value parameter and the data passed by the caller is *copied* into that space. From that point on, a value parameter becomes a variable that is indistiguishable for other variables local to that subprogram.

Note that the above applies for *any* parameter that is passed. For example, if you declare an array of 500 strings of 100 characters each, Pascal actually copies *all 50,000-plus bytes!* Needless to say, this is a time-consuming practice, something to be used only when necessary. For a more efficient method, though one with other consequences, see the discussion of variable parameters.

On the other hand, value parameters are usually a good choice for simpler variable types, since Pascal tends to generate more efficient code when accessing such parameters are. In addition, the fact that such parameters are considered local variables has the effect of isolating them from the caller. Take a look at the example on the next page.

**EXAMPLE:**

```
PROGRAM Parameter_Demo_1 ;

VAR v1 : Integer ;

PROCEDURE Test_1 ( v1 : Integer ) ;
  BEGIN
    WriteLn( 'Test entry: ', v1 ) ;
    v1 := 9 ;
    WriteLn( 'Test exit:  ', v1 )
  END ;

BEGIN
  v1 := 5 ;
  WriteLn( 'Demo start: ', v1 ) ;
  Test_1 ( v1 ) ;
  WriteLn( 'Demo exit:  ', v1 )
END .
```

If you enter and compile this program for TOS, link it, and execute it, the results will look something like this:

```
Demo start: 5
Test entry: 5
Test exit:  9
Demo exit:  5
```

As this shows, neither the duplicate name, V1, nor the fact that the parameter is modified within the procedure affects the variable in the calling program.

## VARIABLE PARAMETERS

When the *VAR* keyword *does* appear ahead of a parameter's name in a subprogram header, the parameter is said to be a variable parameter. During program execution, Pascal allocates space only for a *pointer* to that parameter and the *address* of the given data is placed into the designated space. From that point on, a variable parameter is treated as a pointer, although the pointer aspect is transparent to the user. Specifically, the user does *not* use the pointer symbol ( ^ ) to access that parameter.

Note that the above applies for *any* parameter that is passed. For example, if you declare an array of 500 strings of 100 characters each, Pascal only uses *four bytes* of space in the subprogram as a pointer to that array. Needless to say, this is much faster than the time-consuming method used by value parameter arrays. However, there is a penalty to be paid, in that references to array elements, or any other VAR parameters, may take an extra level of indirection. The time is probably not noticeable except in tight loops, though.
**NOTE:** When a subprogram expects a VAR parameter, the caller *MUST* pass a variable as the actual parameter, not any kind of general expression.

The program on the next page demonstrates variable parameters:

```
PROGRAM Parameter_Demo_2 ;

VAR v1 : Integer ;

PROCEDURE Test_2 ( VAR v2 : Integer ) ;
 BEGIN
  WriteLn( 'Test entry: ', v2 ) ;
  v2 := 9 ;
  WriteLn( 'Test exit:  ', v2 )
 END ;

BEGIN
 v1 := 5 ;
 WriteLn( 'Demo start: ', v1 ) ;
 Test_2 ( v1 ) ;
 WriteLn( 'Demo exit:  ', v1 )
END .
```

This program produces results something like this:

```
Demo start: 5
Test entry: 5
Test exit:  9
Demo exit:  9
```

Notice that this time the change to the formal variable parameter V2 affected the value of the caller's actual parameter. Even the fact that different variable names were used made no difference.

## SUBPROGRAMS AS PARAMETERS

The syntax diagram for a parameter indicates that a SUBPROGRAM HEADER may be passed as a parameter to another subprogram. There are sometimes good reasons for wanting to do something like this. Consider a graphing procedure that needs to know what function to graph. By allowing the caller to specify a mathematical function, a single routine could graph a wide range of mathematical formulae. Parts of such a program might look like this:

```
PROCEDURE Graph( FUNCTION f( x : REAL )
                        : REAL ) ;
  VAR y, Angle : REAL ;
  BEGIN
    Angle := 0.0 ;
    REPEAT
      y := f( Angle ) ;
      ... (* graph y vs. Angle *)
      Angle := Angle + 1.0
    UNTIL ( Angle >= 360.0 ) ;
  END ;

FUNCTION Sine( Theta : REAL ) : REAL ;
  ...
FUNCTION Tanh( Alpha : REAL ) : REAL ;

  ...          (* in some other subprogram, then: *)
  Graph( Sine ) ;
  Graph( Tanh ) ;
  ...
```

It's important to note that you can only pass a fixed form of subprogram. In our example we can only pass the *Graph* procedure a function that takes one *REAL* value parameter and returns a *REAL* result. We could *not* pass a function such as the following as a parameter to *Graph* for two reasons: It needs two parameters and it returns the wrong kind of result.

```
FUNCTION Score( High, Low : REAL ) : Integer ;
  ...
```

Note that the parameters of the subprogram that is passed as a parameter must be declared, but they need not be passed in the call. In our *Graph* example, the range of *angle* values was generated by the *Graph* procedure itself.

## STRINGS AS PARAMETERS

Because STRING is a special data type, it has special rules regarding parameter passing. These rules are easy to remember because they make sense for strings:

When a String (constant or variable) is passed to a VALUE String parameter, its length may not be greater than the maximum size specified in the value parameter's formal declaration.

**EXAMPLE:**

```
PROGRAM Demo ;
VAR   Big : String[ 150 ] ;

PROCEDURE Test( S : String[10] ) ;
 BEGIN
WriteLn( 'String passed: ', S )
  END ;

BEGIN
 Big := 'Test?' ;  Test( Big ) ;
  Test( 'Simplicity' ) ;
  Big := '10 chars!!' ;
 Test( Big ) ;
  Big := 'More than 10 characters' ;
  Test( Big ) ;            (* error! *)
  Test( 'a bit too big' ) ; (* error! *)
END .
```

When a String (always a variable) is passed to a VARIABLE String parameter, its DECLARED size (not necessarily length) must be at least as big as the size specified in the variable parameter's formal declaration.

This makes sense. Consider a subprogram such as this:

```
PROCEDURE Oops ( S : String[ 100 ] ) ;
 BEGIN
   S[ 100 ] := '*'
 END ;
```

If the calling routine tried to pass a variable with declared size less than 100, what character would be altered by the assignment, above? Certainly some character or other bizarre memory location outside the caller's string!

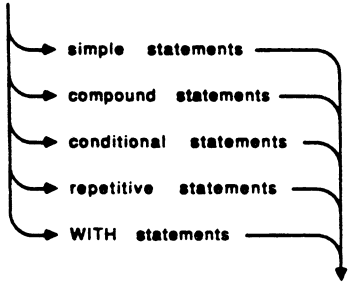
## STRUCTURED TYPES AS PARAMETERS

As noted above, an entire array, and by logical extension, a record, may be passed either as a value or variable parameter.

Similarly, a component of such a structured type, such as an element of an array, or a field of a record may be passed as either a value or variable parameter.

**EXCEPTION:** *Components* of a *PACKED* structure may *not* be passed as variable parameters! This restriction is in the ISO standard because Pascal compilers in general are expected to perform word-sized memory accesses on variable parameters to increase execution speed, and a component of a *PACKED* structure need not be word-sized. An *entire* packed structure *may* be passed, since the declaration of the receiving variable parameter must be of the same *PACKED* type, so the compiler is able to figure out what kind of code to generate.

# STATEMENTS

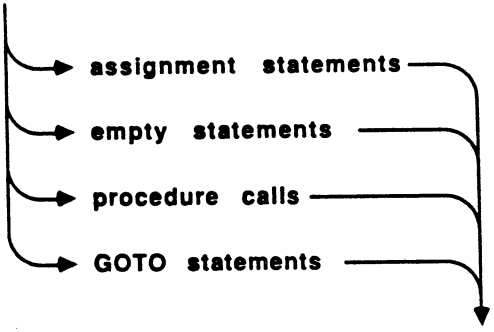



The actions in a Pascal program are called *statements*. *Definitions* and *declarations* don't do anything while your program is running ; they only let you extend Pascal's vocabulary so that the statements in your program can do more complex actions.

Pascal statements are broken into two classes: simple statements and structured statements. Simple statements perform a single action unconditionally. Structured statements determine whether an action should be done instead of blindly going ahead and doing it.

**NOTE:** Semicolons ( ;) are not statement terminators in Pascal, as they are in C. They are statement *separators.* This distinction becomes important when considering structured statements, where a semicolon seems to be terminating a contained statement, when in fact it is separating the *entire structure* from the next statement.

# SIMPLE STATEMENTS




## THE ASSIGNMENT STATEMENT

The most common action in a Pascal program involves assigning a value to a variable, which may be a simple variable, or an element of an array or record. This is done in the assignment statement using the assignment operator :=.

Some examples of assigning values to variables are:

```
num := 253.58e-7 ; {assignment to a Real variable}

last_char := 'Z' ; {assignment to a CHAR variable}

solved := ans=7 ;   {assignment to a Boolean
                     variable}

grid [5,4] := 35 ; {assignment to an array element}

xyz.abc := blue ; {assignment to a record variable }
```

## ASSIGNMENT COMPATIBILITY

When assigning values to variables, make sure that the value is *assignment compatible* with the variable. The *value* is assignment compatible with the *variable* if any one of the following is true:

The variable and the value are of the same type, except for file types and types containing a file type.

The variable is Real, and the value is Short_Integer or Long_Integer.

The variable and the value are of type-compatible ordinal types, including Short_Integer and Long_Integer, and the value falls within the range of the variable's type.

The variable and the value are of type compatible set types, and the value's members are all valid members of the variable's set type.

The variable and the value are type-compatible STRINGs.

**NOTE:** The rules for type compatibility are defined in the section on *TYPES*, page 6-47

**CAUTION:** If Range Checking is turned Off, assignment errors will not occur at run-time.
**SEE:** *Compiler Directives*, page 4-9

## THE EMPTY STATEMENT

The empty statement is very simple. It consists of nothing at all, which is befitting when you consider what it does: nothing. It does require a separating semicolon, however. An empty statement can be used in several situations:

```
IF NOT dead THEN
  CASE weather OF
    Bad  : ; {empty stmt does nothing}
    Fair : MopeAround ;
    Good : HaveFun
  END
ELSE ;       {empty stmt between ELSE and ;}
```

Most instances of the empty statement go unnoticed by the programmer because they are used for syntactic consistency, as in the example above. There is at least one programming application of it, however: when you want nothing done for specific values in a CASE statement, as in the first example above.

## PROCEDURE CALLS

As noted in the section on Subprogram Declarations, Pascal supports two kinds of subprograms: functions and procedures. The former are used in expressions, wherever a value of the appropriate type is required.

Procedures, however, are simply treated as statements, thus extending the vocabulary of the language. Pascal has several required subprograms, automatically supplied by the compiler, and Personal Pascal supplies several additional ones ( both *"built-in" and included via GEMSUBS.PAS and AUXSUBS.PAS* ).

To call a procedure, simply code the name of the desired routine. If the subprogram requires parameters, the procedure name must be followed by a list of parameters (each of the appropriate type) enclosed by parentheses. Some procedure parameters must be variables (and they are so indicated in the descriptions of those procedures later in this manual). All other parameters may be expressions of the appropriate type.

**EXAMPLES:**

```
WriteLn( 'This is sample procedure call.' ) ;
New( Mail_Record_Ptr ) ;
ReWrite( Printer, 'LST:' ) ;
```

## THE GOTO STATEMENT

The *GOTO* statement allows you to make unstructured branches in Pascal. It causes the control flow to go to the statement specified by the *GOTO* label. If this flow change involves scope change, *all* variables *below* the level of the target statement, including *FILE* variables, are de-allocated.

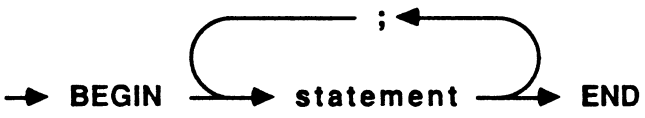
An example:

```
    IF regular_member THEN GOTO 1 ;
    write ( 'Associate Member ' ) ;
  1: writeln ( name ) ;
```

The labelled code in the example will be executed every time the segment runs. Labelling a line doesn't cause it to be skipped. If regular member is true, however, the line, *write ( 'Associate Member ' ) ;* will be skipped.

The GOTO statement defeats the structured programming style associated with Pascal, but is necessary in a few cases. For example, you can exit a program when you are several scope levels deep by using GOTO to reach the end of the program.

## THE COMPOUND STATEMENT

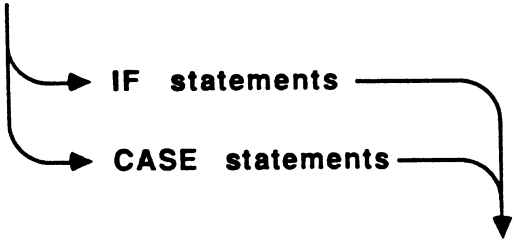



Some structured statements execute a single Pascal action based upon some condition. One action isn't usually enough to do anything useful, so most program operations involve several Pascal statements. The compound statement lets you combine several Pascal statements into one larger action so that other structured statements will execute them as a single group. An example of the compound statement is:

```
BEGIN
  Writeln ;
  FOR i = 1 TO 10 DO
    Write(i) ;
  Writeln('I can count') ;
  Writeln('(at least to ten)')
END ;
```

All of the statements between the BEGIN and END are considered one statement and can be used as such in other structured statements.

# CONDITIONAL STATEMENTS

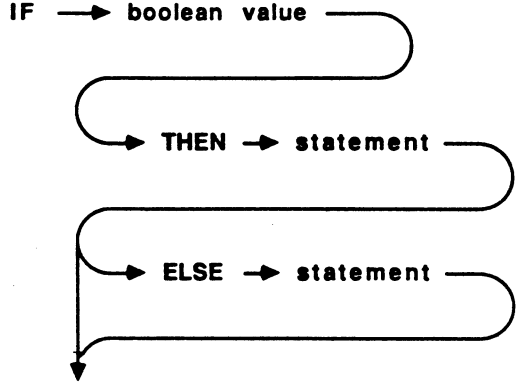



Conditional statements are those that execute actions once, non-repetitively, if a specified condition is met. Personal Pascal supports two such statements:

**IF** page 6-111

**CASE** page 6-113

## THE IF STATEMENT




The *IF* statement allows you to execute an action based on the results of a Boolean expression. It also allows for either-this-or-that situations through the use of the *ELSE* option. The statement following *THEN* is executed if the expression is True. If the expression is False, program control passes to the statement following the IF statement, unless the *ELSE* option is used, in which case the statement following *ELSE* is executed before control passes to the next statement.

Some examples of the IF statement are:

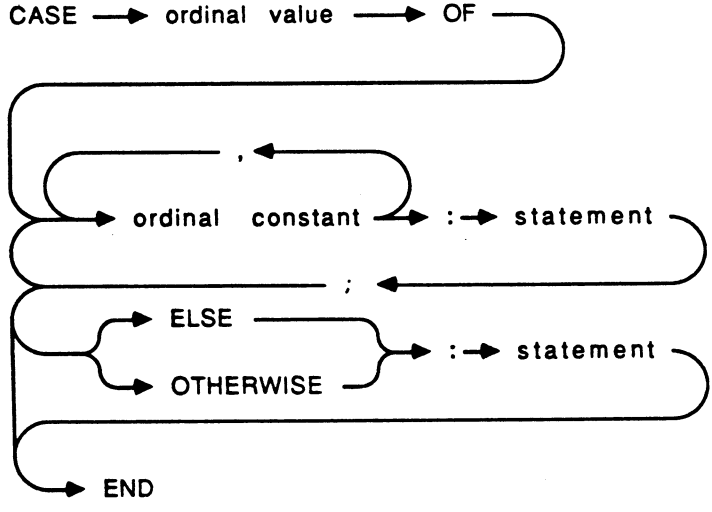
```
IF ok THEN
  DoThis ;

IF retired THEN
  IssuePension
ELSE
  IssueSalary ;
```

**NOTE:** Because the semicolon is a statement separator in Pascal, this

```
IF retired THEN
  Issue_Pension ;
ELSE
  Issue_Salary ;
```

will cause an error, because the compiler sees the semicolon after *Issue_Pension* as the end of the *IF* statement ; when it comes upon the *ELSE* the compiler sees it as an illegal use of *ELSE.*

## THE CASE STATEMENT



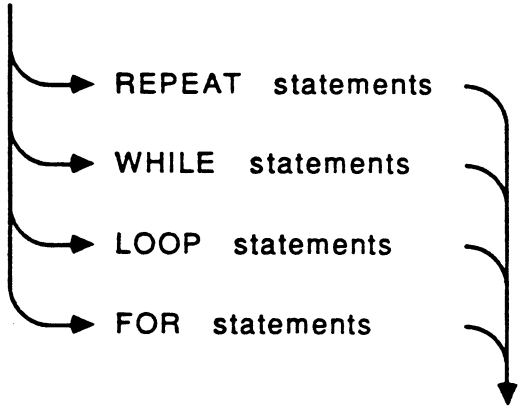

Where IF uses a Boolean expression to determine whether an action should take place, the CASE statement uses an ordinal expression to determine which of several actions to execute. The result of expression must be of an ORDINAL type, and the case values must be of the SAME ordinal type. The ELSE and OTHERWISE options allow you to combine values that are not explicitly handled into a single group. If you want several statements executed as part of a case-item, combine them into a compound statement.

An example of the CASE statement:

```
CASE note OF
  C1,C2 : Writeln('Do') ;
  D1    : Writeln('Re') ;
  E1    : Writeln('Mi') ;
  F1    : Writeln('Fa') ;
  G1    : Writeln('So') ;
  A1    : Writeln('La') ;
  B1    : Begin
            Writeln('Ti')  ;
            Writeln('For') ;
            Writeln('Two')
          End ;
  OTHERWISE : Writeln('Sorry, that's out of my
                                range.')
END ;
```

**NOTE:** No two case value lists may be equivalent or contain the same case value. Because *ELSE* and *OTHERWISE* perform the same syntactic function, you may only have one of them *(1 ELSE or 1 OTHERWISE, not both )* in a single *CASE* statement.

# REPETITIVE STATEMENTS




Repetitive statements allow you to repeat an action. The number of times the action is repeated can be fixed or based on some condition, depending upon which repetitive statement you use. Personal Pascal supports four repetitive statements:

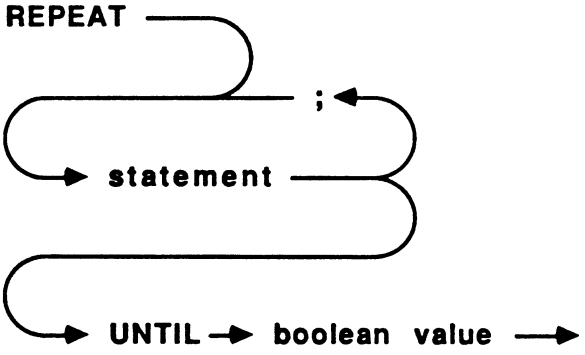
**REPEAT** page 6-116

**WHILE** page 6-117

**LOOP** page 6-118

**FOR** page 6-119

## THE REPEAT STATEMENT



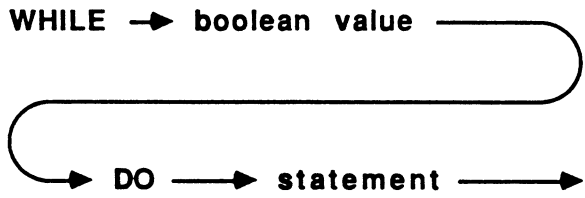

The REPEAT statement allows you to repeat an action until a specified Boolean condition becomes TRUE. An example of the REPEAT statement:

```
REPEAT
  ShowChoices ;
  choice := GetChoice ;
  DoChoice ( choice )
UNTIL choice=Quit ;
```

**NOTE:** Multiple statements between *REPEAT* and *UNTIL* do not need to be grouped into a compound statement because the statement is completed by the *UNTIL* expression.

**ALSO:** If the boolean expression never becomes True, the *REPEAT* will continue forever.

## THE WHILE STATEMENT

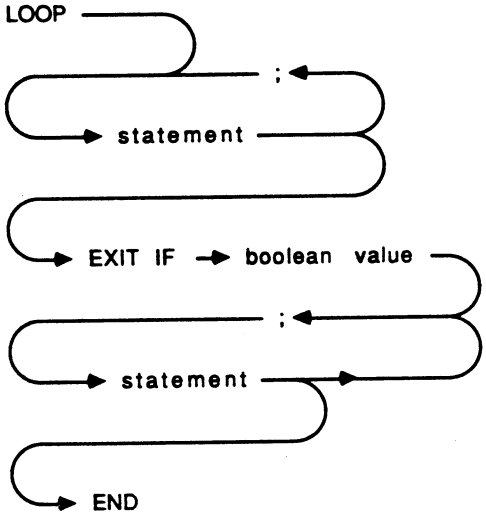



The *WHILE* statement is just the opposite of the *REPEAT* statement because it allows you to repeat an action until a specified Boolean condition becomes *False.* An example of the *WHILE* statement:

```
WHILE month>=1 AND month=<12 DO
  BEGIN
    ShowCalendar ( month ) ;
    month := AskMonth ;
  END ;
```

**NOTE:** If the boolean expression never becomes *False*, *WHILE* will continue forever.

## THE LOOP STATEMENT

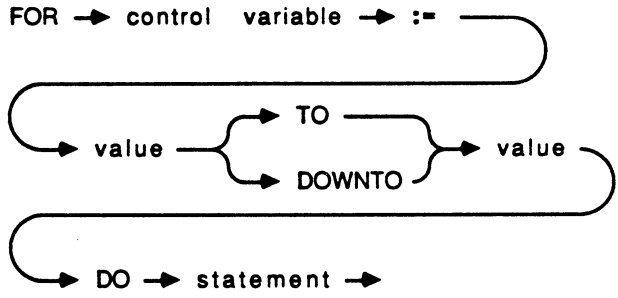



The *LOOP* statement in Personal Pascal offers conditional repetition which can be tested in the middle of the loop. An example:

```
LOOP
  date := GetDate
  EXIT IF BadDate ( date ) ;
  Add_Appointment ( date )
END ;
```

**NOTE:** Multiple statements between *LOOP* and *END* do not need to be grouped into a compound statement because the statement is completed by *END.*

## THE FOR STATEMENT




The FOR statement allows you to repeat an action a specified number of times. It does this by using an ordinal control variable, and initial and final values for that control variable. The control variable must be of an ORDINAL type. Real variables may NOT be used. The initial value and final value must be of the same ordinal type as the control variable.

Some examples of the FOR statement:

```
VAR hue : Rainbow ;
    luminance : 0..9 ;

FOR hue := Red TO Violet DO
  BEGIN
    FOR luminance := 0 TO 9 DO
      DrawStripe ( hue , luminance ) ;
    FOR luminance := 9 DOWNTO 0 DO
      DrawStripe ( hue , luminance ) ;
  END ;
```

The *TO* and *DOWNTO* options determine how the control variable is modified. *TO* causes the control variable to increase by one each repetition, and *DOWNTO* causes it to decrease by one each repetition. If the ordinal type is non-numeric, *By one* means *Successor* in the case of *TO*, and *Predecessor* in the case of *DOWNTO.*

There are some rules regarding the control variable used in a *FOR* statement:

The control variable must be declared in the block that immediately contains the *FOR* statement.

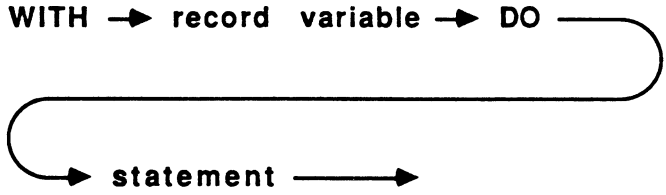
The control variable must be of a simple ordinal type. It cannot be an ordinal part of a structured variable.

The value of the control variable may not be changed within the *FOR* statement, either by direct assignment or by using it as a *VARiable* parameter.

After the *FOR* statement has completed execution, the value of the control variable is undefined.

**NOTE:** When using *TO,* the *FOR* statement will not execute the specified statement if the initial value is greater than the final value. When using *DOWNTO*, it will not execute the statement if the initial value is less than the final value.

## THE WITH STATEMENT




You will often need to make assignments to several fields of a record variable, one after the other, as in:

```
recvar.field1 := value1 ;
recvar.field2 := value2 ;
recvar.field3 := value3 ;
recvar.field4 := value4 ;
recvar.field5 := value5 ;
```

The *WITH* statement allows you to specify a record variable once, and then use just the field names. This can make your code much more readable. Putting the above example into a *WITH* statement would result in:

```
WITH recvar DO
  BEGIN
    field1 := value1 ;
    field2 := value2 ;
    field3 := value3 ;
    field4 := value4 ;
    field5 := value5 ;
  END ;
```

This example isn't much clearer than the original, but when the record variable is more complex the *WITH* statement saves time and increases comprehension:

```
WITH transarray[x].date DO

  BEGIN
    mon := curmonth ;
    day := curday ;
    yr := curyear ;
  END ;
```

replaces

```
transarray[x].date.mon := curmonth ;
transarray[x].date.day := curday ;
transarray[x].date.yr := curyear ;
```

**NOTE:** Because the record variable in the *WITH* statement is evaluated before the statement section is executed, actions that would normally affect the record variable do not change it. In the above example, changing the value of x would normally change the record variable, but it does not have this effect within the region of the *WITH* statement.

# SPECIAL TOPICS

This section covers topics that are not properly part of the definition of Pascal, but that are necesssary to understand in order to use the Personal Pascal implementation of the language in the ATARI ST computer environment.

# FILES AND DEVICES

In Standard Pascal, the *Rewrite* and *Reset* procedures are supposed to bind a file identifier to a physical file, opening it and making it active. ( This file identifier is sometimes called a file variable or file pointer, but both of those are misnomers, since they neither can be varied nor pointed. )

Rewrite is used when the file is for *output*, and Reset is used when the file is for *input*. In standard Pascal, all non-temporary files must be specified in the file identifier list in the program header. On large machine batch systems, the operating system allows the computer operator to link these file identifiers with physical devices or files. TOS has no standardized way to perform such file linking. Thus, as noted in the Program Header section, Personal Pascal ignores any file identifiers given in the file header. This is fairly consistent with other microcomputer Pascals. ( For example; Turbo Pascal )

Instead, Personal Pascal allows a second parameter to the REWRITE and RESET procedures, and that second parameter is intended to be the physical file name.

*Do not confuse the file identifier with the physical file name.* A statement such as this is perfectly legal, although more than a little confusing:

```
Reset( InFile, 'OUTPUT' ) ;
```

The file named *OUTPUT* is presumed to be a disk file on the current active drive, in the current active directory and so might be more properly named *A:\WORKDIR\OUTPUT* or such.

You should *always* specify a file name the first time you use Reset or ReWrite with a particular file identifier in a program or subprogram! However, once a file variable has been bound to a physical file, the second parameter is not necessary to Reset or ReWrite the file identifier again, unless you want to bind it to a different physical file. A logical thing to do is ReWrite a temporary file, giving it a name, and then Reset that same identifier in order to process the information.

In most cases the physical file will be a valid GEMDOS file name, possibly including the drive name or perhaps specifying a complete path name, but the following devices are also supported as physical files:

**'CON:'** The ST console (keyboard and monitor); A valid device for both input and output.

**'AXI:'** Input from the auxiliary (RS232) port; valid for input only.

**'AXO:'** Output to the auxiliary (RS232) port; valid for output only.

**'LST:'** The list device (printer); valid for output only.

**'PRN:'** The same as 'LST:'.

**'NUL:'** A null device that simply discards all output; valid for output only.

As a specific example, consider this method of sending data to a printer:

```
PROGRAM PrintDemo ;
VAR
  Out : TEXT ;

BEGIN
  ReWrite( Out, 'LST:' ) ;
  WriteLn( 'This line goes to screen' );
  WriteLn( Out, 'This one goes to the printer' ) ;
END .
```

Notice that the device name MUST include the terminating colon. If we had coded 'LST' instead, we would have send the line to a disk file named LST in the current default directory. The file name may be given as a constant string, a string variable, or a packed character array.

**COMMENT:** Turbo Pascal uses two procedure calls to accomplish what Personal Pascal does with its modified versions of ReWrite and Reset. The equivalent of Personal Pascal's

```
    Reset( Test, 'A:TEMPDATA' ) ;
```

is Turbo Pascal's

```
    Assign( Test, 'A:TEMPDATA' ) ;
    Reset( Test ) ;
```

This information is provided simply because so many demonstration programs now available are written in Turbo Pascal.

## REDIRECTING SCREEN OUTPUT

Whenever you use a Write or WriteLn call, the data you wish to write is automatically sent to the ST's screen via the VT52 terminal emulator, as discussed in Predefined Subprograms. This is accomplished because the predeclared file identifier OUTPUT is automatically opened to the console, CON:, exactly as if you had coded this statement at the beginning of your program:

```
REWRITE( Output, 'CON:' ) ;
```

The reason this works is because Standard Pascal specifies that all output via Write and WriteLn that does NOT specify a file identifier MUST go to the predefined OUTPUT identifier.

There are at least two cases where you might want to change where this default OUTPUT goes:

If you are debugging a program and would like to have a printed copy of what is flashing by too fast to read on the screen.

If you are writing a GEM program, where Write and WriteLn to the screen must generally be avoided. In this case, having OUTPUT hooked to the printer simplifies your source code a little.

So,if you want to send ALL output from your Write and Writeln statements to the printer, all you have to do is insert the following line somewhere before your program's first use of those statements:

```
ReWrite( Output, 'LST:' );
```

From this point on, all output that normally would show on the ST display will go to the printer.

If you want to switch back to the display after sending output to the printer, use the following statement before the Write or Writeln statements you want sent to the screen:

```
ReWrite( Output, 'CON:' );
```

There is no reason that the default output of Write and WriteLn must go to either screen or printer. You may use ANY legal file or device name, as defined above, for the redirection.

# POINTERS IN PASCAL

In Pascal, pointers may *only* be used to access data that have been dynamically allocated. In particular, standard Pascal provides no way to find the address of any existing global or local variable. While there are ways to cheat and work around this restriction, as shown by some of the demonstration routines on the Personal Pasal disks, they are *not* guaranteed to work. In any case, the likelihood of their being transportable to other versions of Pascal is remote at best.

Because the use of pointers is so restricted, we must first discuss *how* Pascal allocates and deallocates dynamic data areas. Further, though the two routines that accomplish these tasks are called *PROCEDURES*, they violate standard procedure parameter rules so much that it may be more apt to think of them as special kinds of statements.

## NEW and DISPOSE

New ( p ) ;
Dispose ( p ) ;

The New procedure allocates a dynamic variable of p's "domain-type" so that the dynamic variable can be accessed using the variable access form p^. If p is a pointer to some type of object, then that particular type is its domain type.

**NOTE:** The value of the dynamic variable is undefined after it is allocated. Do *NOT* count on memory being cleared, strings being given zero length, etc.

The Dispose procedure *de-allocates* the dynamic variable associated with the pointer p, leaving p undefined. In Personal Pascal, *de-allocates* simply means that the space used by the dynamic variable is returned to the heap so it can be allocated again later.

## REFERENCING POINTER VARIABLES

Pointers may be compared to other pointers of the same domain-type, or the constant NIL, using the relational operators "=" and "<>". But no other comparisons are valid for pointers.

A pointer variable may be legally *modified* in one of only four ways. For each of the ways below, assume that the following declaration is in effect (and that each statement is executed in the order given):

```
VAR  p,r : ^INTEGER ;
```

You may assign any pointer the the predefined pointer constant NIL.

```
p := NIL ;
```

You may allocate dynamic data storage and assign it to a pointer via the NEW procedure.

```
NEW ( p ) ;
```

You may assign it the value of another pointer of the same domain-type.

```
r := p ;
```

At this point, you may use either "r" or "p" to refer to the data area obtained via the NEW call. For example:

```
p^ := 29 ;
WriteLn( r^ ) ;
```

This will output the expected results, just as if you had coded

```
WriteLn( 29 ) ;
```

You may use the DISPOSE procedure to deallocate the data storage referenced by a pointer.

```
DISPOSE( r ) ;
```

**CAUTION:** It is an error in standard Pascal to use the data pointed to by a variable that has been "disposed" of, but no method of indicating this error is specified. Virtually no Pasal compiler could keep track of all pointer variables that refer to the same allocated space. With Personal Pascal, you might get a *"pointer out of heap"* error if you now attempted this statement:

```
WriteLn( p^ ) ;
```

But if some subsequent NEW had allocated that same space to some other pointer, the results are unpredictable.

# DYNAMIC ALLOCATION OF VARIANT RECORDS

When you use the NEW procedure to allocate dynamic variables of a variant record type, you can specify which variants you want active by adding additional parameters to the New and Dispose procedures. This saves large amounts of heap space when the variants are of greatly differing sizes.

```
New ( p, c1, c2 ... cN ) ;
Dispose ( p, c1, c2 ... cN ) ;
```

The variable p is a pointer with a variant record domain-type. Constants c1 through cN are case constants in successively deeper levels of variance for the variant record type.

For example, consider these declarations:

```
TYPE
  vtype = RECORD
    fixed : Real ;
    CASE Boolean OF
      True : ( vs : String[ 40 ] );
      False: ( vp : ^vtype );
    END ;
VAR
  vrec : ^vtype ;
```

Given the example, a call to

```
NEW( vrec, True ) ;
```

would allocate 46 bytes of dynamic memory and make it accessible via vrec^, but a call to

```
NEW( vrec, False ) ;
```

would allocate only 10 bytes for use by vrec^.

Once the dynamic variant record variable is allocated, it's subject to the same restrictions as normally allocated variant records, but with some dangers. After a call for the *FALSE* variant of VREC, any attempt to access VREC^.VS may cause an error or unpleasant side effects. A statement that invalidly accesses the *INACTIVE* variant, such as

```
vrec^.vs := 'A simple test' ;
```

could wipe out unpredictable areas of memory outside the dynamically allocated space.

Finally, a major restriction is that you *MUST* Dispose of the variable in *EXACTLY* the same way that you allocated. System crash is *virtually guaranteed* if you perform this pair of calls in sequence:

```
NEW( vrec,True ) ;
DISPOSE( vrec,False ) ;
```

# STRINGS: THE INSIDE INFO

Personal Pascal strings can be seen as packed arrays of characters. A declaration such as:

```
VAR
  st : String[40] ;
```

shares many characteristics with this declaration:

```
VAR
  pa : PACKED ARRAY [0..40] OF CHAR
```

The declaration of a string variable establishes a maximum size to which it can grow, but a string also has an *active* length associated with it. For example, it's legal to make an assignment such as:

```
  st := 'Only 18 characters' ;
```

after which the length of *st* will be 18. Personal Pascal supplies a *Length* function that determines a string's current length.

NOTE: An assignment such as this is *illegal*:

```
  pa := 'Fewer than 40 characters' ;
```

because *pa* isn't a *string!*

Personal Pascal stores the current length of a string in the zero (0th) position of the pseudo-array. That means we could find the length of "st" this way:

```
Len_of_st := ORD( st[ 0 ] ) ;
```

Which serves to point out another feature of strings: Individual characters are accessible. This can have strange effects, as in this sequence:

```
st := 'ABCDE' ;
st[ 12 ] := CHR( 48 ) ;
WriteLn( st ) ;
```

All of that is legal, but the results of the WriteLn will be simply "ABCDE" and neither the twelfth character nor any intervening characters will be output. On the other hand, following the above with this can produce other strange results:

```
st[ 0 ] := CHR( 20 ) ;
WriteLn( st ) ;
```

Twenty characters will be output. We know what the 12th and the 1st through 5th characters are, but the remaining characters will be whatever occupied st's current memory space before this sequence. Because there's no way to write a procedure to handle strings of varying declared sizes as VAR parameters, Personal Pascal supplies predefined subprograms to make string manipulation easier.
**SEE:** *Predefined Subprograms*, page 6-145.

# PERSONAL PASCAL BIT MANIPULATION

Standard Pascal does not specify any bit manipulation capabilities, since it must be implemented on a variety of machines utilizing markedly different architectures. Personal Pascal, however, provides for the more important bit manipulations.

We noted the bit manipulation operators in the subsection on Expressions: they are the bitwise-and ( & ) and the bitwise-or ( | ). They work the same way that similar operators do in other languages. On a bit-by-bit basis the following rules apply:

| | |
|---|---|
| 0 \| 0 is 0 | 0 & 0 is 0 |
| 0 \| 1 is 1 | 0 & 1 is 0 |
| 1 \| 0 is 1 | 1 & 0 is 0 |
| 1 \| 1 is 1 | 1 & 1 is 1 |

**EXAMPLES**:

```
WriteLn( '$', ($3FA & $555) :4:h );
```
which should produce $0150

```
WriteLn( '$', ($3FA | $555) :4:h );
```
which should produce $07FF

In addition to these operators, you may use *SHL* and *SHR*, described as Bit Manipulation Functions in the Predefined Subprograms subsection on page 6-.

# SETS

Pascal's implementation of the *SET* type is unusual in microcomputer languages. Essentially, you are allowed to have a collection of zero or more objects, each belonging to a particular declared type. You can add to or subtract from this set, or use several other operations, and Pascal keeps track of which objects the set contains. Note that a set can not contain more than one of any given object. For example, let's pretend we are going to ask our program's user to choose one or more letters of the alphabet. Then, given these declarations:

```
TYPE
  Upper_Case = 'A..'Z ;
  Letters    = SET OF Upper_Case ;
VAR
  Choices : Letters ;
```

We might make an initial assignment of

```
  Choices := [ ] ;
```

which assigns the empty set to our variable. Then we might add additional elements to our set via code such as this:

```
  Choices := Choices + [ 'Q ]
```

In practice, sets are implemented internally by assigning a data field where each bit of the field represents one possible element of the set. In our example above, then, Pascal would assign 26 bits to *Choices*. The assignment of the empty set would clear all the bits. The addition of element *Q* would turn on the 17th bit and, due to the way the 68000 processor in the ST works, this set would occupy a long word, 32 bits, equivalent to a Pascal Long_Integer, in memory.

For extended information on set manipulation, we must recommend a good tutorial or generic Pascal reference manual, such as the ones by Doug Cooper, mentioned elsewhere in this manual.

## SET CONSTRUCTORS

This is a very brief overview of set constructor notation. In the above example, we used two set constructors: [ ] and [ 'Q ]. A set constructor looks very much like an ordinal type declaration enclosed in square brackets. Some examples of valid set constructors are:

```
[] { The empty set }
['a'..'z','A'..'Z'] { base-type CHAR }
[1,2,3,5,7,11,13] { base-type Short_Integer }
```

Or, slightly more complicated:

```
TYPE
  SeasonType = (Winter, Spring, Summer, Fall) ;
VAR
  Seasons : SeasonType ;
  ...
  Seasons := [ Winter..Summer ] ;
```

These constructors are NOT legal:

```
[ 5..3 ] ;
[ 'T'..'H' ] ;
```

Nor is this one, assuming the example shown above:

```
Seasons := [ Fall..Spring ] ;
```

**NOTE:** The empty set is the only constructor that is compatible with all set base-types.

# ASSEMBLY LANGUAGE SUBROUTINES

When you write external assembly language programs to interface with Personal Pascal, you must abide by the following rules:

## REGISTERS

You may use the data registers D0 through D6 and address registers A0 through A5 without saving their previous contents. If you use D7 or A6, save their contents before using them, and restore them before returning to the Pascal program.

## PARAMETERS

Parameters are pushed onto the stack in the same order that they're declared in the formal parameter list. If the parameter is VARiable, a 32-bit long word which is the address of the actual parameter is pushed onto the stack. If the parameter is VALUE, the entire parameter is pushed onto the stack. We do not recommend this choice for anything except simple data types; Integers, Reals, etc..

For value STRINGs, as many words are pushed onto the stack as needed to store a string of the maximum DECLARED length, NOT the current length. Again, we do not recommend this choice.

When FUNCTIONs or PROCEDUREs are used as parameters to other subprograms, including assembly subprograms, they occupy two long words. The first one is the static link address, which must be loaded into register A0 before the subprogram is called. The second is the execution address of the subprogram. We do not recommend passing subprograms as parameters to an assembly language subprogram to any except the most advanced programmers.

**NOTE:** Your assembly language routine must pull ALL the parameters from the stack before returning to the Pascal program.

## FUNCTION RETURN VALUE:

Assembly language and Pascal FUNCTIONs return their values in data registers. Simple data values, except Long_Integer and Real, are returned in D0.W. Long_Integer and pointer values are returned in D0.L. The four most significant bytes of a Real value's mantissa are returned in D0.L, and the least significant byte of the mantissa and the scale factor are returned in D1.W.

**SEE:** The example function on the next page.

This example performs a bitwise AND of two Long_Integer values. Assume the following 68000 assembly language:

```
BITAND
  MOVE.L (A7)+,D3 ; pop off return addr
  MOVE.L (A7)+,D0 ; pop off 2nd parameter
  MOVE.L (A7)+,D1 ; and the 1st parameter
  AND.L  D1,D0  ; and values, result in D0
  MOVE.L D3,-(A7) ; push return addr back on
  RTS  ; back to Pascal, value in D0
```

The Pascal declaration of this routine might be:

```
FUNCTION BITAND( v1,v2 : Long_Integer )
      : Long_Integer ;
      EXTERNAL ;
```

You could now write a Pascal program including:

```
  WriteLn( BITAND( $7F3,$1A ) );
```

**CAUTION:** In theory, any assembler can be used to write the assembly code. In *practice*, you must be able to *link* the Pascal object code and assembler output. Personal Pascal's linker supports version 2's unique format AND the Atari-standard DRI format. If you are not using a DRI-compatible assembler, you will need a program to convert object file formats or a linker capable of handling various object file formats. Unfortunately, most assemblers in the ST market do *NOT* follow the proper Atari object format.

# PREDEFINED SUBPROGRAMS

As previously noted in the description of Subprograms, standard Pascal specifies that several subprograms *must* be part of any Pascal package. Although some of these required subprograms could be written in Pascal, and are, others simply cannot be due to restrictions within the definition of the language. In particular, some required subprograms accept either varying *numbers* of parameters, as Write and Read, or various *types* of parameters, as *PRED* and *SUCC*. These subprograms *must* be specially recognized by the compiler so that it can generate appropriate, and possibly varying, code.

Personal Pascal version 2 includes 3 classes of Predefined Subprograms:

Those required by the Pascal *standard.* In this section they are designated by the word "STANDARD."

Those that are *specific* to the *Personal Pascal* compiler.

Those considered to be *auxiliary* subprograms, for optional use in Personal Pascal programs. Designated by the words "NEEDS AUXSUB.PAS."

The first two of these require no additional work on your part of beyond using them in a program. The third class, though, requires you to include a declaration file, *AUXSUBS.PAS,* in the declarations to your program.

## USING AUXILIARY SUBPROGRAMS

All subprograms in this class are clearly designated in the following descriptions. To use these them, you must include this file from your Personal Pascal A disk in the program:

AUXSUBS.PAS

You may copy this file into the appropriate spot in your program or use Personal Pascal's INCLUDE directive using a format such as this:

```
PROGRAM Your_Choice_of_Name ;
   { $I AUXSUBS.PAS }
   { Put your declarations here }
BEGIN
   { Main program code }
END.
```

**NOTE:** If your program will use both some of the auxiliary subprograms and some of the GEM subprograms described in the next section of this manual, you must include (via the {$I... directive) BOTH files. For example, your program might start with

```
PROGRAM A_GEM_Program ;

  {$I GEMSUBS.PAS }
  {$I AUXSUBS.PAS }
  ... (* etc. *)
```

The pages that follow describe the various predefined subprograms in more detail. The subprograms are grouped by type.

# BOOLEAN FUNCTION

In Pascal, most of the functions that return a Boolean value are used in special situations and operate on specific data types. The only "general purpose" Boolean function is the ODD function.

**ODD ()** **Standard**

```
b := Odd ( n ) ;
```

n A Short_Integer or Long_Integer.
b A Boolean.

The Odd function returns *TRUE* if the number n is odd, and *FALSE* if it's even.

# ORDINAL FUNCTIONS

Excepting for CHR, the ordinal functions are needed in Pascal to determine the NUMBERING of an ordinal value. Although these functions may be used with any ordinal type, they are most useful with enumerated types and character types.

For the examples of ordinal functions shown below, assume that these declarations have been made:

```
TYPE
  Seasons = (Winter, Spring, Summer, Fall) ;
  Color_Wheel =
    (Red, Orange, Yellow, Green, Blue, Violet) ;
```

# ORDINAL FUNCTIONS

**ORD ()** **Standard**

```
i := Ord ( o ) ;
```

o Any Ordinal data type.
i An Integer.

Ord returns the integer index value of o in its base type. Examples:

```
Ord('a')        is  97
Ord(True)       is   1
Ord(False)      is   0
Ord(27)         is  27
Ord(Summer)     is   2
Ord(Green)      is   3

Ord(2.95)       is illegal because 2.95 is a real
                   number and is not ordinal
```

**PRED ()** **Standard**
**SUCC ()** **Standard**

```
o2 := Pred ( o1 ) ;
o2 := Succ ( o1 ) ;
```

o1 Any Ordinal data type.
o2 Same type as o1.

These two functions allow sequential movement through ordinal types. PRED returns the element immediately preceding the argument in the argument's data type, and SUCC returns the element immediately following the argument in its own data type.

**EXAMPLES:**

```
Pred(True)     is  False
Succ(False)    is  True

Pred('b')      is  'a'
Succ('b')      is  'c'

Pred(5)        is  4
Succ(5)        is  6

Succ(Spring)   is  Summer
Pred(Spring)   is  Winter

Succ(True)     produces an error since True is the
               last Boolean element.
```

## CHR ()

**Standard**

c := Chr ( n ) ;

n An Short_Integer or Long_Integer.

c A Char.

CHR is a unique function that returns the character represented by the ASCII code of its argument, hence n must be in the range 0..255 whether it is an Short_Integer or Long_Integer.
NOTE: A SET of CHAR may only have 128 elements.

**EXAMPLES:**

```
Chr(97)  is 'a'
Chr(27)  is the ESCAPE character
Chr(13)  is the RETURN character
Chr($33) is '3'
```

# TRANSFER FUNCTIONS

Standard Pascal provides functions that convert different types of simple data. Personal Pascal adds some to deal with its two available Integer types.

**ROUND ()** **Standard**
**TRUNC ()** **Standard**

```
i := Round ( r ) ;
i := Trunc ( r ) ;
```

r A Real.
i An Integer.

These two functions are used to change Real data into Integer data. Round returns the integer value nearest r, while Trunc returns the integer portion of r, cuting off any fractional digits.

**EXAMPLES:**

Round(2.7) is 3
Trunc(2.7) is 2

Round(-3.1) is -3
Trunc(-3.1) is -3

Round(2.35) is 2
Trunc(2.35) is 2

ROUND and TRUNC are actually implemented in Personal Pascal by substituting calls to Long_Round and Long_Trunc or Short_Round and Short_Trunc, depending on whether the compiler's default to Long_Integer directive, ( {$I+} ) has been used.
**SEE:** the following pages for more on this topic.

**LONG_ROUND ()**
**LONG_TRUNC ()**

```
i := Long_Round ( r ) ;
i := Long_Trunc ( r ) ;
```

r A Real.

i A Long_Integer.

These functions convert Real data into Long_Integer data. The programmer may code these functions as shown or may set the compiler's {$I+} directive, in which case ROUND and TRUNC are synonyms for these functions.

## SHORT_ROUND ()
## SHORT_TRUNC ()

```
i := Short_Round ( r ) ;
i := Short_Trunc ( r ) ;
```

r  A Real.
i  A Short_Integer.

These functions convert Real data into Short_Integer data. The programmer may code these functions as shown or may set the compiler's {$I-} directive, in which case ROUND and TRUNC are synonyms for these functions. If no directive is used to specify the default integer size, then Short_Integer is assumed, so that ROUND and TRUNC are synonyms for these directives.

## INT ()

```
i := Int ( l ) ;
```

l A Long_Integer.
i An Integer.

This function converts Long_Integer data into Short_Integer data.

These examples assume that "L" is a Long_Integer variable, "S" is a Short_Integer variable:

```
L := 232000 - 200000 ;
S := Int( L ) ;
{ S will have the value 32000 }

L := $1FFFF - $10000
S := Int( S ) ;
{ S will have the value $FFFF,
which is same as -1, decimal }
```

**NOTE:** *INT* is optional in version 2 of Personal pascal, but is retained for compatibility with version 1.

# ARITHMETIC FUNCTIONS

These functions may be used with either Integer or Real arguments. Since they correspond to common mathematical operations, they are grouped as Arithmetic Functions.

**ABS ()** **Standard**

n2 := Abs ( n1 ) ;

n1 An Integer, Long_Integer, or Real.
n2 Same type as n1.

Abs returns the absolute value of the number n1.

**EXAMPLES:**

Abs(4) is 4 (Short_Integer)
Abs(-45000) is 45000 (Long_Integer)
Abs(-2.42E-8) is 2.42E-8 (Real)

**ARCTAN ()** **Standard**
**COS ()** **Standard**
**SIN ()** **Standard**

```
r2 := ArcTan ( r1 ) ;
r2 := Cos ( r1 ) ;
r2 := Sin ( r1 ) ;
```

r2 A Real.
r1 A Real.

These three functions perform the common trigonometric operations *arctangent, cosine*, and *sine*, respectively. They require *Real* parameters and return *Real* values.

**NOTE:** These functions *ALWAYS* assume that angles are measured in radians.

Example:

```
  WriteLn( Sin( 0.78539816 ) :9:6 ) ;
```

should give a result of 0.707107

NOTE: 0.78539816 is approximately 45 degrees.

**EXP ()** **Standard**
**LN ()** **Standard**

```
r2 := Exp ( r1 ) ;
r2 := Ln ( r1 ) ;
```

r1 A Real.
r2 A Real.

These two functions perform the operations "natural exponent" and "natural logarithm", respectively.

That is,

Exp(1.0) is approximately 2.718281828
Ln(2.718281828) is approximately 1.0

**PWROFTEN ()**

```
r := PwrOfTen ( i ) ;
```

i A Short_Integer in the range 0..38.
r A Real.

The PwrOfTen function returns the Real value of 10.0 raised to the power given by the argument. This function is probably only useful in doing numeric conversions.

**SQR ()** **Standard**
**SQRT ()** **Standard**

```
n2 := Sqr ( n1 ) ;
r2 := Sqrt ( r1 ) ;
```

n1 A Short_Integer, Long_Integer, or Real.
n2 Always the same type as n1.
r1 a Real.
r2 a Real.

The SQR function returns the value of the square of its argument. It usually provides a marginal speed improvement over coding n1*n1, but may not be as easy to read.

The Sqrt function returns the square root of its argument, but the returned value is ALWAYS of type Real.

**EXAMPLE:**

Sqrt( Sqr( 7 ) ) is 7.0 (not Integer 7!)

## SHL ()
## SHR ()

```
n := ShL ( val, bits ) ;
n := ShR ( val, bits ) ;
```

| | |
|---|---|
| val | A Short_Integer or Long_Integer. |
| bits | A Short_Integer or Long_Integer in the range 0..31. |
| n | Same type as val. |

These two functions return the value that results when the first argument is bit-shifted the number of times specified by the second argument. SHL performs a left shift, and SHR performs a right shift, and both operations are done logically (i.e. without regard to the sign of the first argument).

**EXAMPLES:**

```
Rsh( $FFFF, 3 )  is $1FFF
Rsh( -1, 3 )     is $1FFF
```

(Because -1 decimal is the same as $FFFF, and the sign is ignored in these bit shiifts.)

```
Lsh( $555555,1 ) is $AAAAAA
Rsh( $8421, 1 )  is $4210
```

(Notice that the low order bit is lost.)

# ARRAY TRANSLATIONS

The predefined procedures *PACK* and *UNPACK* allow you to pack an unpacked array for better storage and then unpack it so its elements can be used as *VARiable* parameters.

The advantage of using a *PACKED ARRAY* is that it saves storage space. But remember that the Pascal Standard does not allow elements of a *PACKED* type to be used as *VAR* parameters, so the disadvantage of packing an array is that its elements may not then be passed as a *VARiable* parameter until it is unpacked, and unpacking slows your program.

In any case, in Personal Pascal, the only array element types that benefit (in terms of memory usage) from a *PACKED* declaration are those ordinal types that may be contained in a single byte ( including the standard types *BYTE* and *CHAR* ).

**PACK ()** **Standard**

Pack ( vunpacked, start, vpacked ) ;

| | |
|---|---|
| vunpacke | An unpacked array variable. |
| start | A value of the index type of vunpacked. |
| vpacked | A packed array with the same component type as vunpacked. |

The Pack procedure will copy specified components of the unpacked array (first argument) into a packed array (third argument), packing them in the process.

The components of the unpacked array that will be copied begin with the array element specified by the second argument and continue to the last element of that array. A run-time error will occur if the packed array given is too small for the number of components to be copied.

**NOTE:** *PACK* will only pack one-dimensional arrays.

**UNPACK ()** **Standard**

UnPack ( vpacked, vunpacked, start ) ;

| | |
|---|---|
| vpacked | A packed array variable. |
| vunpacked | An unpacked array with the same component type as *vpacked.* |
| start | A value of the index type of *vunpacked.* |

The *UNPACK* procedure operates similar to Pack, except it reverse. It copies elements of the packed array into the unpacked one, starting at the element specified. A run-time error will occur if the unpacked array is too small for the number of elements to be copied.

**NOTE:** UnPack will only unpack one-dimensional arrays.

# STRING MANAGEMENT SUBPROGRAMS

Personal Pascal provides six subprograms that perform common text operations on STRING data. But, as noted in the Special Topics subsection on Strings, the user may easily access the individual characters of strings and thus produce other routines. Some of these predefined string subprograms are provided because they "know" information about a string (such as its declared size) that a user written routine can not determine. Other routines are provided simply for convenience.

## LENGTH ()

i := Length ()

len := Length ( str ) ;

len  An Integer.
str  A String variable.

The Length function returns the current length of its string argument, and it is useful when manipulating string variables character by character, as in this example that "uppercases" a string:

```
FOR i := 1 TO Length(str) DO
  BEGIN
    IF str[i] IN ['a'..'z'] THEN
      BEGIN
        j := Ord(str[i]) - Ord('a')+ Ord('A') ;
        str[i] := Chr(j) ;
      END ;
  END ;
```

## CONCAT ()

i := Concat ()

dest := Concat ( s1, s2, ..., sN ) ;

| | |
|---|---|
| dest | A STRING variable. |
| s1, s2, sN | String constants, or *STRING* , *CHAR*, or *PACKED ARRAY OF CHAR* variables. |

The Concat function will concatenate a variable number of parameters and return the concatenated string. The parameters are concatenated in the order of their appearance in the parameter list. A run-time error will result if the destination string is not large enough to accommodate the concatenated string. (This is an example of a routine that could not be written as-is in Pascal.)

Example:

```
s1 := 'easy' ;
s2 := 'strings' ;
s1 := Concat( s1, ' ', s2 ) ;
```

s1 will now contain the string
'easy strings'

## COPY ()

str := Copy ()

dest := Copy ( source , start, size ) ;

| | |
|---|---|
| dest | A STRING variable. |
| source | A STRING variable. String constants are illegal. |
| start | A Short_Integer in the range 1 to 255. |
| size | A Short_Integer in the range 0 to 255. |

The *Copy* function will copy a substring of one string into another string. The substring begins at the start character of the source string. The size parameter is the length of the substring in characters. A run-time error will result if the given size is greater than the length of the source or if the destination string variable is too small to contain the substring.

**NOTE:** This function is similar to the *MID$* function of many implementations of *BASIC.*

## POS ()

i := Pos ()

locator := Pos ( pattern, str ) ;

| | |
|---|---|
| locator | An Integer. |
| pattern | A string constant, or STRING , Char, or PACKED ARRAY OF CHAR variable. |
| str | A STRING variable. |

The *POS* function searches the given string argument for the designated pattern. If the pattern is found, POS will return its starting position in the string. If the pattern is not found, POS will return a zero.

**EXAMPLE:**

Pos( 'the', 'this is the test' ) ;

would return a position of 9.

## INSERT ()
## DELETE ()

Insert ( source, dest, start ) ;
Delete ( str, start, size ) ;

| | |
|---|---|
| source | A string constant, or STRING , Char, or PACKED ARRAY OF Char. |
| dest | A STRING variable. |
| start | A Short_Integer in the range 1 to 255. |
| str | A STRING variable. Constants are illegal. |
| size | A Short_Integer; 0 to 255 inclusive. |

The *INSERT* procedure inserts the entire source string into the destination string beginning at the specified starting position (in the destination). A run-time error results if the resultant string would be larger than the declared length of the destination.

The *DELETE* procedure will remove a substring of a given size from a string variable beginning with the specified starting character. A run-time error will occur if the specified substring runs beyond the end of the string variable.

**C_TO_PSTR ()** **Needs AUXSUBS.PAS**
**P_TO_CSTR ()** **Needs AUXSUBS.PAS**

C_To_Pstr ( Cstr, Pstr )
P_To_Cstr ( Pstr, Cstr )

Pstr A String variable
Cstr A C_String variable ( C_String is declared in the file AUXTYPES.PAS )

**NOTE:** For the call to C_To_Pstr, Pstr must be declared as a maximum size string. That is, as String[255].

These procedures convert between the Personal Pascal string format and the 'C' language format expected by many operating system calls. In Pascal, strings are represented as 1 dimensional arrays in which element 0 contains the string's length, and the remainder of the array contains its characters. A 'C' format string contains characters, beginning with element 0, and has a nul [i.e., CHR(0) ] byte as its last element.

# FILE SUBPROGRAMS

There are two kinds of file operations supported by Personal Pascal: Those that manipulate data to and from files, and those that manipulate whole files.

## FILE I/O SUBPROGRAMS

Before any data can be transferred from or to a file, the file must be opened. That is, a Pascal file identifier must be associated with a particular device or file. Since the actual implementation of this action is highly system-dependent, we refer you to the *FILES* topic in the *SPECIAL TOPICS* subsection.

After a file is opened, and depending upon the mode of opening, you may read data from it or write data to it. Finally, you must close the file to ensure that the file is intact on whatever device is in use.

Only the last of these steps is automatic: Pascal automatically closes a given file as it exits the block where the file identifier associated with it is declared. This must be either the same block or a caller of the block where the file is opened.

**REWRITE ()** **Standard**
**RESET ()** **Standard**

```
Rewrite ( f, name ) ;
Reset ( f, name ) ;

ReWrite ( f ) ;
Reset ( f ) ;
```

| | |
|---|---|
| f | A FILE type variable. |
| name | A string constant, STRING, or PACKED ARRAY OF CHAR that is a valid TOS device or file name. |

*Rewrite* and *Reset* bind a file identifier to a physical file, opening it and making it active. The first form shown, with a name given to the device or file being opened, is preferred in Personal Pascal, though it does NOT conform to standard Pascal.

The second form, though conforming to standard Pascal, will NOT work in most cases.

ReWrite is used when the file is to be used for output.

Reset is used when the file is to be used for input.

**IMPORTANT:** Read the FILES topic in the SPECIAL TOPICS section before attempting to use ReWrite or Reset.

**NOTE:** The name given for the physical file must be a valid GEMDOS device name or file name, such as:

```
ReWrite( printer, 'LST:' ) ;
Reset( datafile, 'B:\DATADIR\TEMPFILE' ) ;
```

**CAUTION:** If you REWRITE a disk file that already exists, the original file is erased. You MAY send output to a file which has been opened with Reset. This allows you to update existing files.

**ALSO:** If you try to Reset a non-existent disk file no error will occur, but Eof will become True unless you've modified IO_Check and are checking your own errors.

## EOF ()  Standard

```
done := Eof ( f ) ;
```

done  A Boolean.
f  Any FILE identifier.

The Boolean function *EOF* returns *True* if the file designated by the given identifier is empty beyond what is currently available in the buffer via the pointer expression f^. Otherwise, it returns *False.*

**GET ()** **Standard**
**PUT ()** **Standard**

```
Get ( t ) ;     (* Standard form *)
Put ( t ) ;

Get ( f, n ) ;  (* Random access form *)
Put ( f, n ) ;
```

t Any FILE identifier.
f Any FILE identifier, except one of type TEXT.
n A Short_Integer or Long_Integer.

The *GET* and *PUT* procedures are the basic forms of data transfer from and to files. GET retrieves the next component from the file bound to the specified identifier and stores it in the associated buffer. PUT writes the current contents of the buffer to the file bound to the specified identifier. The contents of the buffer are accessed by treating the file identifier as a buffer pointer, as shown on the next page:

```
TYPE
  MLType = RECORD
    name    : String[25] ;
    address : String[80] ;
    mlcode  : Char ;
    balance : Real ;
  END ;
VAR
  List     : MLType ;
  Listfile : FILE of MLTYPE ;
  Amtfile  : FILE of Real ;
 ...
Reset( Amtfile, 'B:AMOUNTS' );
ReWrite( Listfile, 'A:DEBTORS' );
 ...
REPEAT
  ...
   List.balance := List.balance + Amtfile ^ ;
   Listfile ^  := List ;
   Put( Listfile ) ;
   Get( Amtfile ) ;
  ...
```

Here, we read from Amtfile and add (" Amtfile ^") to List.balance, then the Listfile record is written and Amtfile is read again.

**CAUTION:** When you Reset a disk file, the buffer will contain the FIRST COMPONENT of the file. Use GET AFTER the use of the pointer to the current record of Amtfile. When you Reset another device, the buffer variable is undefined and must be filled using Get.

## RANDOM ACCESS

The ISO Pascal Standard provides only for sequential access to files. But Personal Pascal extends the procedures Put and Get to provide random access to file components.

To allow such random access, Personal Pascal adds an optional second parameter to both PUT and GET. This parameter may be a Short_Integer or Long_Integer, and is the index of the component to be transferred. If you consider a file as if it were an array of its component type, this index would be equivalent to the array's subscript.

**NOTE:** The index of the first component of a file is always zero, not one.

**NOTE:** Random access may NOT be used with TEXT type files because they don't have a rigidly fixed component type or size.

**READ ()** **Standard**

Read ( f, v1, v2, ..., vN ) ;

f Any FILE type variable.
v Variables of the component type of f.

The *READ* procedure allows you to input an arbitrary number of components directly into variables. The statement

```
Read(f,v1,v2) ;
```

is equivalent to

```
BEGIN
  v1 := f^ ;
  Get( f ) ;
  v2 := f^ ;
  Get( f ) ;
END ;
```

The buffer variable acts as a look-ahead area for the file identifier; it contains the next component to be accessed, not the one currently being accessed.

If a file variable is not specified as the first *READ* parameter, the predefined *TEXT* variable *INPUT* is assumed. For more on accessing *INPUT* and other TEXT files, see the special topic on *TEXT* files later in this section.

## WRITE ()  Standard

Write ( v1, v2, ..., vN ) ;

f  Any FILE type variable.
v  Variables of the component type of f.

The *WRITE* procedure allows you to output an arbitrary number of expressions to a file. The expressions must all be assignment compatible with the component type of the given file identifier. The statement:

```
Write(f,v1,v2) ;
```

is the same as

```
BEGIN
  f^ := v1 ;
  Put( f ) ;
  f^ := v2 ;
  Put( f ) ;
END ;
```

If a file variable is not specified as the first *WRITE* parameter, the predefined *TEXT* variable *OUTPUT* is assumed. *TEXT* files are treated separately in a later topic of this section.

## SEEK ()

Seek ( f, n ) ;

f Any FILE type variable, except Text.
n A Short_Integer or Long_Integer.

The ISO Pascal Standard provides only for sequential access to files. Aside from the extensions to *PUT* and *GET*, Personal Pascal also provides the *SEEK* procedure.

*SEEK* moves TOS's internal file pointer so that the next transfer to or from the physical file bound to the given file identifier will access the specified component n of that file. No data transfer is performed.

**NOTE:** Recall that the first element of a file is always numbered as *zero*.

## CLOSE ()

Close ( f ) ;

f Any FILE type variable.

The Close procedure closes the specified file identifier, making further access of its data illegal. This procedure is provided because TOS has an ill-defined upper limit on how many files may be open at once. Since Pascal's automatic file closing is clumsy, at best, this procedure provides a convenient way to release a file when your Pascal program is done processing its data.

# TEXT FILES

Standard Pascal specifies and requires one predefined file type and two predefined file identifiers. TEXT is the predefined type, and for some, but not all purposes, it is functionally identical to *FILE OF CHAR.*

The proper declaration is:

```
VAR
  printer : TEXT ;
```

You should NOT code *"FILE OF TEXT"*. The compiler will likely toss its cookies if you try.

The two predefined file identifiers are INPUT and OUTPUT. Type TEXT is assumed. In fact, you can code this without changing default operation at all:

```
VAR
  Input, Output : TEXT ;
```

By default, at the start of a program, both these identifiers are bound to the 'CON:' device; the ST's keyboard is used for INPUT, and the TOS text screen, not a GEM window, is used for output. These lines wouldn't affect the default operation of your program:

```
Reset(   Input, 'CON:' ) ;
ReWrite( Output, 'CON:' ) ;
```

**NOTE:** As noted in the "FILES" topic in the "Special Topics" subsection, you can bind INPUT and OUTPUT to other devices or files (using Reset and Rewrite). The "FILES" topic gives an example of using Rewrite with OUTPUT so that normal screen output goes to the printer instead.

**NOTE:** There are several predefined subprograms in Personal Pascal that may be used ONLY with TEXT files. Also, the procedures READ and WRITE perform differently when used on Text files.

## EOLN ()  Standard

done := Eoln ( t ) ;

done  A Boolean.
t  A TEXT identifier.

The EOLN function returns FALSE until the next character in the file is the end-of-line. This is useful when treating a TEXT file as single characters rather than as line-oriented text data, though the use of a String variable with ReadLn may be more useful.

## PAGE ()  Standard

Page ( t ) ;

t  A TEXT identifier.

The PAGE Procedure outputs an end-of-page character to the specified TEXT file. On the ST, this is equivalent to:

```
Write( t, Chr(12) ) ;
```

Using this procedure instead tends to make your programs more portable to other Pascal implementations.

**READ ()** **Standard**
**READLN ()** **Standard**

Read ( textfile, v1, v2 ... vN ) ;
Readln ( textfile,v1,v2 ... vN ) ;

**NOTE:** For use of READ with files other than those of type TEXT, see the discussion on page 6-178.

These two procedures allow you to input various types of simple data from a Text file. The TEXTFILE identifier may be omitted if you are reading from INPUT. The variables v1 through vN can be any of the predefined simple types except Boolean. Only the last variable in the list may be a STRING. Depending on the data type of the variable, one of the following will occur:

**Short_Integer or Long_Integer:** The input processor skips over all spaces and end-of-line characters. It then reads the longest sequence of characters that form a signed decimal number. The first non-digit character terminates the sequence. This non-digit is the first character a subsequent input will encounter. The processor then attempts to convert this digit-sequence into an integral number.

**Real:** The input processor starts out as though it were looking for a Short_Integer or Long_Integer, but inspects the first non-digit. If it is a period, the processor accepts more digits as the fractional part of the Real. If the first non-digit, or the first non-digit after the fractional part, is an "E", the processor gets a signed scale factor. The processor then attempts to convert this character sequence into a Real value.

**Char:** Since Text files are nominally PACKED FILE OF CHAR, no special transformations re done, except that the end-of-line character is converted into a space. Only one character is extracted from the input stream.

**String:** When the LAST variable is a String, the processor puts all characters up to the next end-of-line into the String variable.

The difference between Read and Readln comes into effect AFTER the input of a variable: Readln eats up all the characters up to and including the next end-of-line character, so that the next access will start at the beginning of the next line. You can ignore a whole line of input text from file t with this statement:

```
Readln(t);
```

For data-sensitive applications, we strongly recommend reading entire lines into String variables and then parsing the strings.

**WRITE ()** **Standard**
**WRITELN ()** **Standard**

Write( textfile, p1, p2, ...pN );
WriteLn( textfile, p1, p2, ...pN );

**NOTE:** For use of WRITE with files other than those of type TEXT, see separation discussion, above.

These two procedures allow you to output various types of simple data to a Text file. The "textfile" identifier may be omitted if you are writing to the predeclared identifier OUTPUT. The parameters p1 through pN designate both the data to be written as well as (optionally) the format in which it is to be written. The general form of each parameter is:

expression : width : special

The actual data that will be written depends on both the value AND type of the expression. Whether the "special" sub-parameter in any given parameter is usable (and what it means) depends on the type of the expression. In any case, though, the width sub-parameter, if given, specifies the MINIMUM number of characters to be output. We consider this minimum number to be a FIELD SIZE, and the ASCII representation of the expression is right justified within this field which is padded with spaces if necessary. The SPECIAL sub-parameter may not be coded unless the sub-parameter field is also coded.

Each of the data types usable with this form of Write and WriteLN is discussed separately below. Unless otherwise noted, no SPECIAL field may be used with a given type.

**CHARACTER EXPRESSIONS:** A single character is output. If a width is given, the character is placed in the rightmost position of the designated field. If no width is given, only the character is output.

**INTEGERS** ( Short_Integer and Long_Integer ): The ASCII representation of the decimal value of the integer is output. If the field width given is too small to accept all needed digits, including a minus sign, if necessary, then all the digits are output anyway. No Pascal error occurs though the appearance of the output may be altered unpleasantly. If no width is given, the default width for Short_Integers is 6; for Long_Integers, 11.

**FOR INTEGERS ONLY:** if a special subparameter of :H or :h is coded, the number is output in hexadecimal notation, rather than decimal.
**SEE:** the discussion of bit manipulation in the *Special Topics* section, page 6-138.

**REALS:** By default, Reals are output in "E" notation ( e.g.: 7.33124E+06 ). Only if both width AND SPECIAL sub-parameters are given will a fixed point notation be used, and then only if the real expression is in range, so that the given width and SPECIAL can accomodate it. For Reals, the special sub-parameter designates the number of digits to the right of the decimal point that should be displayed. For example:

```
WriteLn( 7.335471E+02 : 12 : 2 ) ;
```

will produce something like this: 733.55

**STRINGS:** If no width is given, the entire string is output. If the string is longer than the given width, it is cut off so that only the appropriate leftmost characters are written. If the string is shorter than the given width, it is padded on the left with blanks.

**BOOLEAN:** A boolean expression evaluating as *TRUE* is treated exactly as if the string 'true' had been used as the expression. The width rules are the same as for strings. Similarly, a boolean expression evaluating to *FALSE* is treated as the string 'false'.

**NOTE:** The WIDTH and SPECIAL sub-parameters are Short_Integers and may be coded as expressions if desired. This may be especially handy for formatting reports, etc.

**ERASE ()**
**RENAME ()**

Erase ( f1 ) ;
Rename ( f1, f2 ) ;

f1 Any FILE identifier that is bound to a disk file.
f2 An identifier of the same data type as f1, also bound to a disk file.

These two routines belong to the second class of file routines discussed in the intro to the predefined I/O subprograms. These operations work on entire files and don't care about the data contained in them.

The Erase procedure closes and then erases the physical disk file bound to the specified file identifier.

The Rename procedure closes the file bound to the identifier given by the first argument, closes and erases the physical file bound to the identifier given by the second argument, and then renames the first physical file to the name of the erased file.

Because these routines work with only file identifiers, they must be preceded by appropriate RESET or REWRITE calls. For this reason, the equivalent GEMDOS routines may be easier to use.

**IO_CHECK ()** **Needs AUXSUBS.PAS**
**IO_STATE ()** **Needs AUXSUBS.PAS**

```
IO_Check ( b ) ;
b := IO_State ;
```

b A Boolean.

Finally, these routines may be used with *any* of the I/O subprograms discussed above.

The IO_Check procedure allows you to control whether I/O errors generate run-time errors that terminate program execution. When you pass TRUE as a parameter, I/O checking is performed. (This is the state at program initialization.) As a consequence, any I/O errors encountered generate fatal errors, presenting the user with an error Alert message.

When a program passes *False* as the parameter to IO_Check, it can then detect and handle its own I/O errors.

In a complex program, it may not be convenient to use a global variable to remember the state of the IO_Check flag. Thus, the IO_State function performs this task for you. It simply returns *TRUE* or *FALSE,* according to the last call to IO_Check.

## IO_RESULT Needs AUXSUBS.PAS

```
i := IO_Result ;
```

i A Short_Integer.

This function, also, may be used with any of the above described I/O routines.

When I/O checking is off, after a call of the form:

```
IO_Check(FALSE)
```

a program can determine the status of an I/O operation by using the IO_Result function. Several return values are possible:

IO_Result returns zero if no error occurred with the most recent I/O operation.

If TOS generates an error, IO_Result returns a value returned which is a TOS error number. These are always negative numbers as shown on the next page:

## TOS ERROR CODES

-1 Fundamental TOS system error
-2 Drive not ready
-3 Unknown error
-4 CRC error
-5 Bad request
-6 Seek error
-7 Unknown medium
-8 Sector not found
-9 No paper
-10 Write fault
-11 Read fault
-12 General error
-13 Write protect
-14 Medium change
-15 Unknown device
-16 Bad sectors on format
-17 Disk change
-32 Invalid function number
-33 File not found
-34 Path not found
-35 Too many open files (no handles left)
-36 Access denied
-39 Insufficient memory
-40 Invalid memory block address
-46 Invalid drive
-49 No more files
-64 Range error
-65 Internal error
-66 Invalid program load format
-67 Setblock failure due to growth restrictions

## PASCAL ERROR CODES

If Pascal's I/O processor generated the error, IO_Result will return a positive number:

1 Input past end of file
2 Reset required prior to input
3 Rewrite required prior to output
4 Random I/O attempt to a device
5 Negative random access record number
6 Reset or Rewrite required before random I/O
7 Bad digit encountered in number Read
8 Overflow when Reading number
9 Bad file name in Reset or Rewrite
10 Bad digit encountered in Real Read
11 Error during Real Read

# POINTER MANAGEMENT SUBPROGRAMS

The two subprograms most important to pointer management were already introduced under the POINTERS topic in the SPECIAL TOPICS subsection.

In Pascal, the only legal way to acquire memory space to be accessed by a pointer is to use the procedure NEW with a pointer argument. Then, when the space is no longer needed, the space can be released via a call to DISPOSE.

One consideration: All dynamic memory space in Personal Pascal is allocated from the heap. Remember that the size of the heap and stack combined is controllable by the compiler directives {$Snn} and {$Unn}. If your Atari ST has a megabyte of memory but you use a {$S20} directive, then your Personal Pascal program will have only 20K bytes of stack and heap space. On the other hand, recall that if neither of these directives is coded then Personal Pascal uses virtually all of unused memory for its stack and heap.

Besides NEW and DISPOSE, Personal Pascal provides two procedures that help keep the heap space clean.

**MARK ()**
**RELEASE ()**

```
Mark ( p ) ;
Release ( p ) ;
```

Each of these procedures takes a single pointer parameter of any domain type. Mark stores the current end-of-heap address in the pointer passed to it, and Release sets the end-of-heap address to the address in the pointer passed to it.

These two procedures are useful when you GOTO out of a scope because dynamic variables are not de-allocated by the GOTO. By MARKing the heap's end when you enter the scope where the label is defined, and then RELEASEing the heap when you exit, you effectively de-allocate all dynamic variables allocated within that scope or any subset of it.

**CAUTION:** Many of the subprograms in Personal Pascal's GEM library use NEW to acquire buffers, etc., for their working space. (In particular, and as examples only, New_Menu, New_Dialog, and Set_DText use this technique.) You should NOT surround a block of code that uses such GEM routines with MARK and RELEASE.

# MISCELLANEOUS SUBPROGRAMS

## BASEPAGE ()

BasePage ( p ) ;

The BasePage procedure takes one pointer parameter of any domain type and stores the address of TOS's base page in it. The pointer may then be used to extract specific TOS information like the command line, TPA address, etc. ( See: TOS manual from Atari Developers' Kit or equivalent ).

## OPTION ()

b := Option ( arg ) ;

b A Boolean.

arg - A character string constant, STRING variable, or PACKED ARRAY OF CHAR variable.

Option tests whether its argument is one of the arguments on the TOS command line, returning TRUE if it is, or FALSE if it's not.

**EXAMPLE:**

```
IF Option('/Format')=False THEN
  Writeln('Formatter not installed');
```

## CHAIN ()

Chain ( cmdln ) ;

cmdln — A character string constant, STRING variable, or PACKED ARRAY OF Char

Chain exits the currently running program, passing cmdln to TOS as a command line. Passing the command line allows you to chain one program to another. When the Chained program is done, control returns to the program that made the Chain call.

**EXAMPLE:** Chain( 'drawpic.prg' ) ;

**CAUTION:** When using CHAIN:

Chain does not Close files, so you must make sure none are active before Chaining.

You must use either the {$S} or {$U} compiler directive to reserve enough memory for the chained-to program.

When Chaining from a GEM program, you must close and delete any windows you own, erase any menu bar you're using, and exit from GEM before Chaining.

For TOS programs, it is a good idea to turn off the cursor before calling CHAIN or before returning from a CHAINed-to program.

## CMD_ARGS
## CMD_GETARG ()

```
i := Cmd_Args ;
Cmd_GetArg ( n , arg ) ;
```

| | |
|---|---|
| i | A Short_Integer variable. |
| n | A Short_Integer expression. |
| arg | A String variable. |

Cmd_Args returns the number of arguments currently in the TOS command line buffer.

Cmd_GetArg assigns the TOS command line argument number specified by the first argument to the VAR parameter (a string variable) specified by the second argument. If the requested argument number is less than one or greater than the number of command line arguments, the string argument is set to a length of zero.

Example:

```
FOR i = 1 TO Cmd_Args DO
 BEGIN
  Cmd_GetArg( i, request ) ;
  WriteLn( 'Command arg number ',i );
  WriteLn( '    is ',request );
 END ;
```

## CLOCK

```
I := Clock ;
```

I A Long_Integer.

Clock returns a Long_Integer representing the number of seconds that have elapsed since midnight of the current system date.

**NOTE:** The number of seconds will always be an even number due to the resolution of the ST system timer.

| | |
|---|---|
| **GET_TIME ()** | **Need AUXSUBS.PAS** |
| **GET_DATE ()** | **Need AUXSUBS.PAS** |
| **SET_TIME ()** | **Need AUXSUBS.PAS** |
| **SET_DATE ()** | **Need AUXSUBS.PAS** |

Get_Time( VAR hour,minute, second: Short_Integer )
Get_Date( VAR month, day, year: Short_Integer )

Set_Time( hour, minute, second: Short_Integer )
Set_Date( month, day, year: Short_Integer )

These procedures get and set the time and date. Time is specified in the system 24 hour format. If you specify an odd number of seconds when setting the time, SetTime rounds down to the next lower even number of seconds to match the ST's clock requirement.

Note that the two "GET" routines require variables as parameters, since the information is passed back in those variables.

## FILENAME ()

```
b := Filename ( str ) ;
```

b  A Boolean.
str  A STRING with declared length of at least 80.

FILENAME tests the given string argument to see if it is a valid TOS file name, returning True if it is and False if it isn't.

**EXAMPLE:**

```
IF Filename(fstr)=False THEN
  Writeln('Illegal file name') ;
```

## HALT

```
Halt ;
```

Halt terminates the program currently running after Closing all files, and is most useful to stop a program if an error occurs.

## HANDLE ()

```
i := Handle ( f ) ;
```

i An Integer.
f A FILE identifier of any type.

Handle returns the TOS handle of the given file variable. This handle, not the Pascal FILE variable, may then be used to make direct GEMDOS calls.

**EXAMPLE:**

```
FUNCTION physbase : Long_Integer ;
  XBIOS( 2 ) ;

PROCEDURE blkread( hndl : Short_Integer ;
                   size, locn : Long_Integer ) ;
  GEMDOS( $3F ) ;
  ...
  ...
  Reset( fp, 'SCRNDUMP' ) ;
  blkread( handle(fp), 32000, physbase ) ;
  (* given above declarations,
    last two lines will read date
    from file directly into current screen memory  *)
```

## KEYPRESS

b := KeyPress ;

b A Boolean.

FUNCTION KeyPress : Boolean ;

KeyPress returns a Boolean that becomes True when a keyboard key is pressed.

**EXAMPLE:**

```
VAR c : Char ;
...
...
REPEAT
  Writeln('Press any key to exit.')
UNTIL KeyPress ;
Read( c ) ;
```

**NOTE:** You MUST read the key to clear the flag.

## MEMAVAIL

l := MemAvail ;

l A Long_Integer.

FUNCTION MemAvail : Long_Integer ;

MemAvail returns the amount of memory left, in words, between the top of the stack and the end of the heap. The stack allocates memory from one end of this space when a subprogram is called, and releases it when the subprogram finishes execution. NEW allocates space from the other end of the stack area; the heap. DISPOSE releases the space, returning it to the available memory pool.

**CAUTION:** The MemAvail function returns the true amount of memory only if it is alled before the heap becomes fragmented by NEW requests. Even subsequent DISPOSE calls will not necessarily increase MemAvail, although there may be a significant amount of available memory in the pool.

Example:

```
IF MemAvail<$1000 THEN
 Writeln('Not enough memory');
```

## SIZEOF ()

```
i := SizeOf ( ident ) ;
```

| | |
|---|---|
| i | An Integer. |
| ident | A TYPE or variable identifier. |

SizeOf returns the allocated (or to-be-allocated) size (in bytes) of the specified data type or variable.

**EXAMPLE:**

```
maxblocks := MemAvail DIV SizeOf(block) ;
```

# MACHINE ACCESS SUBPROGRAMS

**SUPER**
**IN_SUPER**

Super ( True )
IF In_Super Then ...

PROCEDURE Super ( YesNo : Boolean ) ;
FUNCTION In_Super : Boolean ;

With a value of TRUE, the Super procedure puts your program into supervisor mode to allow direct unprotected access to the 68000 system. Declaring Super( False ) returns it to the status that existed before you entered supervisor mode. If the program was in supervisor mode before you asserted Super( True ), it will remain in supervisor mode. If not, it will leave supervisor mode.

The IN_SUPER function simply tests whether or not the system is currently in supervisor mode. This can be useful in subroutines, where the state of the supervisor flag needs to be remembered and restored.

## MACHINE ACCESS SUBPROGRAMS

| | |
|---|---|
| **PEEK ()** | **Needs AUXSUBS.PAS** |
| **WPEEK ()** | **Needs AUXSUBS.PAS** |
| **LPEEK ()** | **Needs AUXSUBS.PAS** |
| **POKE ()** | **Needs AUXSUBS.PAS** |
| **WPOKE ()** | **Needs AUXSUBS.PAS** |
| **LPOKE ()** | **Needs AUXSUBS.PAS** |

Peek ( address: Long_Integer): Byte
Wpeek ( address: Long_Integer): Short_Integer
Lpeek ( address: Long_Integer): Long_Integer

Poke ( address: Long_Integer; value: Byte )
Wpoke (address: Long_Integer; value: Short_Integer)
Lpoke ( address: Long_Integer; value: Long_Integer)

The Peek functions return the value stored at a memory location. The Poke procedures put a value into a memory location.

Peek and Poke pass single-byte values, Wpeek and Wpoke pass Short_Integer values, ans Lpeek and Lpoke pass Long_Integer values.

**CAUTION:** Except for PEEK and POKE, the addresses used in these routines must be even numbers.

**MOVE_BYTE ()** **Needs AUXSUBS.PAS**
**MOVE_WORD ()** **Needs AUXSUBS.PAS**
**MOVE_LONG ()** **Needs AUXSUBS.PAS**

Move_Byte ( source, dest, count: Long_Integer)
Move_Word ( source, dest, count: Long_Integer)
Move_Long ( source, dest, count: Long_Integer)

The Move procedures copy a block of memory from one area to another. Source and destination (the first two parameters) are the beginning addresses of the two blocks. Count (the last parameter) is expressed as the number of bytes, words or long words to be moved.

**CAUTION:** These routines do not check the validity of address boundary values. You must do that yourself. Specifically, the source and destination addresses must be even numbers when using Move_Word or Move_Long. Also, if any portion of the move will access protected memory, then you shuld put the processor in supervisor state first.

## ADDRESS FUNCTIONS

The Pascal specification does not provide any way to find the address of a variable, nor does it provide a method to force a pointer to reference a particular memory address.

In the microcomputer environment, though, there are occasionally times when both these capabilities would be handy to have. The Personal Pascal AUXSUBS.PAS and PASAUX files provide some functions that you may use to "cheat" under certain circumstances.

**PTR_BYTE()**
**PTR_CHAR()**
**PTR_INTEGER()**
**PTR_LONG_INTEGER()**

```
FUNCTION Ptr_Byte( where : Long_Integer )
      : B_Ptr ;
FUNCTION Ptr_Char( where : Long_Integer )
      : C_Ptr ;
FUNCTION Ptr_Integer( where : Long_Integer )
      : I_Ptr ;
FUNCTION Ptr_Long_Integer( where : Long_Integer )
      : L_Ptr ;
```

**NOTE**: The function return types shown (B_Ptr, C_Ptr, I_Ptr, L_Ptr) are predefined in AUXSUBS.PAS as ^Byte, ^Char, ^Short_Integer, and ^Long_Integer, respectively. They must be predefined because a function may not be declared as returning a new type.

Each of these functions takes a Long_Integer, which is presumed to be a memory address, as an argument and returns a pointer of the given type which points to that same address. This pointer may then be used to provide direct access to the given memory location.

**CAUTION:** The Long_Integer given as the argument must be a valid address within the ST's memory space. Further, for Ptr_Integer and Ptr_Long_Integer, the address must be an even number. (Violation of either of those conditions will result in a bus error when the pointer is used.) Finally, unless the address is within Pascal's heap space, you must direct the compiler to turn off pointer checking ( via {$P-} ) to avoid a run-time error.

EXAMPLE:

```
VAR
  pport : C_Ptr ;
...
pport := Ptr_Byte( $FFFA2B ) ;
Write( 'Receiver status on 68901 chip is ' ) ;
WriteLn( pport^ ) ;
...
```

**ADDR_BYTE()**
**ADDR_CHAR()**
**ADDR_INTEGER()**
**ADDR_LONG_INTEGER()**

```
FUNCTION Addr_Byte( VAR byte_var : Byte )
         : Long_Integer ;
FUNCTION Addr_Char( VAR char_var : Char )
         : Long_Integer ;
FUNCTION Addr_Integer(
           VAR short_var : Short_Integer )
         : Long_Integer ;
FUNCTION Addr_Long_Integer(
           VAR long_var : Long_Integer )
         : Long_Integer ;
```

You may use these functions to find the memory storage address of any Byte, Char, Short_Integer, or Long_Integer variable. Simply call the appropriate function, passing a variable of the proper type as the parameter, and the return value will be a Long_Integer representation of the address of the parameter.

**EXAMPLE:**

```
VAR
  Lvar  : Long_Integer ;
  Laddr : Long_Integer ;
  Lptr  : L_Ptr ;
...

Laddr := Addr_L( Lvar ) ;
Write( 'Variable "Lvar" is at address $' ) ;
WriteLn( Laddr : 6 : h ) ;
Lptr := Ptr_L( Laddr ) ;
Write( 'Its contents are ') ;
WriteLn( Lptr^ ) ;
Write( 'Which should be the same as ' ) ;
WriteLn( Lvar ) ;
```

**PTR ()**
**ADDR ()**

The eight functions provided, above, are nice to have. But what happens if you need the address of a record? What happens if you need a pointer to an array? The PASAUX file provide two functions that provide generic solutions to this problem, but you will have to provide your own declaration(s).

The PTR and ADDR functions already exist in PASAUX, so they can be declared as "EXTERNAL" in your program. Because of their nature, you may wrap almost any pointer declarations around them that you need. An example will help:

```
TYPE
  Test = RECORD
       vs : Short_Integer ;
       vl : Long_Integer ;
       END ;
  Testptr = ^Test ;
VAR
  Tvar : Test ;
  Tptr : Testptr ;

FUNCTION Addr( VAR what : test )
      : Long_Integer ;
      EXTERNAL ;
FUNCTION Ptr( where : Long_Integer )
      : Testptr ;
      EXTERNAL ;


  ...
  Write( 'Address of "Tvar" is $' ) ;
  WriteLn( Addr( Tvar ) : 6 : h ) ;
  Tptr := Ptr( $430 ) ;
  Write( 'The Long_Integer at $432 contains ') ;
  WriteLn( Tptr^.vl ) ;
```

Simply replace type definitions, above, with ones that match your situation. Only one limitation: You may only declare the Addr and Ptr functions as External once (each) per block. Because of this, if you need more than two kinds of "Pointer" declarations in a program, you may need to build small functions that make local declarations and then return appropriate types.

## FORMATTED STRING CONVERSION

### READV
### WRITEV

ReadV( source, v1, v2, ...vN )
WriteV( dest, p1, p2, ...pN )

source  A String.
dest  A variable, type String[255]

These routines work exactly like *ReadLn* and *WriteLn* except that a string is used in place of a file identifier. Thus these may be used to convert strings to numbers and vice versa. In the case of WriteV, this includes the ability to perform formatted conversions. After using such a conversion, the resultant string may be displayed in a window (via *Draw_String*) or dialog box (via *Set_DText* ).

**EXAMPLE:**

```
WriteV( str, 'Total: $', Balance:13:2 ) ;
Draw_String( x, y, str ) ;
```

# VT-52 OPERATIONS

These PROCEDURES implement the VT-52 terminal commands. As such, they are available ONLY in TOS (non-GEM) programs.

***NOTE:*** **To use these procedures you must include *AUXSUBS.PAS.***

The operations implemented are as compatible as possible to those available to Turbo Pascal, a product and trademark of Borland Int'l.

The first group of PROCEDUREs are available in both Personal Pascal and Turbo Pascal:

**ClrEol**

Clears text from the current cursor position to the end of the current line.

**ClrScrn**

Clears the entire screen of text and graphics and moves the cursor to the top left corner of the screen.

**CrtInit**

No function. This command is included to maintain compatibility with Turbo Pascal source code.

### CrtExit

No function. This command is included to maintain compatibility with Turbo Pascal source code.

### DelLine

Deletes the entire line at the current cursor position and moves the remaining lines up one position.

### InsLine

Inserts a blank line at the current cursor position and moves the current line and those below it down one position. The bottom line of a full screen is lost.

### GotoXY ( x, y )

x is a Short_Integer expression.
y is a Short_Integer expression.

Positions the cursor at the specified character ( x ) and line ( y ) coordinates. The top left corner is 0,0.

### InverseVideo

Causes subsequent printing to be in inverse video, that is, with foreground and background colors switched.

### NormVideo

Causes subsequent printing to be in normal video.

### TextColor ( color )

color is a Short_Integer expression.

Sets foreground text color to the given color register number (0..15). Subsequent text output to the screen will be drawn in this color.

**TextBackground ( color )**

color is a Short_Integer expression

Sets background text color to the given color register number (0..15). Subsequent text output to the screen will be drawn over a background of this color.

The following PROCEDUREs are implemented in Personal Pascal, but not in Turbo Pascal. They are included here to give the programmer easy access to all the VT52 emulator capabilities.

**Curs_Up**

Moves the cursor up one line.

**Curs_Down**

Moves the cursor down one line.

**Curs_Right**

Moves the cursor right one character position.

**Curs_Left**

Moves the cursor left one character position.

**Curs_Home**

Moves the cursor to the top left corner of the screen.

**Curs_Up_2**

Moves the cursor up two lines. At the top of the screen, inserts a blank line.

**Clr_EOS**

Clears text from the current cursor position to the end of the screen.

**Curs_On**

Turns the flashing cursor on.

**Curs_Off**

Turns the flashing cursor off. No cursor is displayed.

**NOTE:** This procedure should be called before leaving a TOS program.

BLANK
PAGE

# THE GEM LIBRARY

Personal Pascal is a natural choice for the Atari ST programmer because of its extensive GEM/Pascal library. GEM is an acronym for Graphics Environment Manager, an invention and trademark of Digital Research. From the user's point of view, GEM is the desktop with its icons, windows and mouse. From the programmer's point of view, GEM is the magic inside the operating system that performs graphics operations. Personal Pascal provides you with a set of functions and procedures to make working with GEM as painless as possible.

There are some features of GEM that the GEM/Pascal library does not implement. They generally duplicate other, easier methods, or they are very difficult to use. In any case, Personal Pascal doesn't EXCLUDE the use of these features; you can make direct calls to GEM's routines using either generic GEM calls or C-type procedures and functions.
**SEE:** *Language Directives.*

Personal Pascal's GEM library should prove to be more than adequate to easily and quickly develop programs using the best features of GEM.

# THE COMPONENTS OF GEM

GEM's primary components are:

**The Menu Bar and its Menu Items**
**Alert Boxes**
**Dialog Boxes**
**Windows**
**The Mouse Pointer**
**The Keyboard**
**Peripheral Devices; Printers, etc.**

**THE MENU BAR** is the line at the top of the screen containing the words *Desk*, *File*, *View*, and *Option*. When you move the mouse pointer to one of these words, a Drop Down Menu appears. Of all the available selections, only the word *Desk* and most of the items in its drop down menu are fixed. The other words in the menu bar, and all the selections under them are placed there by you, the application programmer.

**ALERT BOXES** serve as warning or error messages in the GEM system. To see an example of an alert box, try to drag one of the disk drive icons onto the trash can.

To see a typical **DIALOG BOX**, use the *Files* menu to ask GEM to create a new folder. Dialog boxes are used to pass messages and responses between your program and the user. They cause your program to pause until the user responds to a dialog.

A dialog box can also act as a sort of monolog box. It can display information without waiting for any user response. The compiler uses such a dialog box to keep you informed of its progress.

The boxes on the screen where the file icons appear are **WINDOWS**. Optional features of a window include the scroll bars at the right and bottom sides, the title bar at the top, and the *close*, *grow*, and *full-screen* corner boxes.

The **MOUSE POINTER** has several available forms, including the busy bee icon, cross hairs and two types of hands. You can also make custom pointers.

An important GEM concept is that of **EVENTS**. A GEM event is anything requested by the user or by the GEM system. Events can be highly interactive processes. When the user presses a key or clicks a mouse button, GEM reports the event to your program. Your program can then tell GEM to do something in response. GEM may then ask your program to do something else as a result. For example; if you asked GEM to remove a window from the screen, it might respond by passing your program one or more re-draw requests, asking you to show the parts of other windows that were hidden by the window being removed.

The only GEM functions that don't require event processing are alert boxes and dialog boxes.

# GEM ACCESS

Personal Pascal uses two files to provide easy interface to GEM. They are:

| | |
|---|---|
| **GEMSUBS.PAS** | *a source file* |
| **PASGEM** | *an object file* |

The source file must be included at the proper place in your source code . You should use this general form in your programs:

```
PROGRAM name ;

  {$I GEMSUBS.PAS}

 CONST{ only if you use global constants }
     { your global CONSTant definitions go here }
 TYPE{ only if you use global types }
       { your global TYPE definitions go here }
 VAR  { only if you use global variables }
    { your global VARiable declarations go here }
        { to use Personal Pascal enhancements }

             { your subprograms go here }
 BEGIN { the main body of the program }
   IF Init_Gem >= 0 THEN
     BEGIN
          { your program's main body goes here }
       Exit_Gem ;
     END ;
 END.
```

GEMSUBS.PAS contains predefined constants, types, and external subroutine declarations. In general we will refer to constants and types by name. Their definitions won't be given in the text since the names alone are sufficient when using the GEM/Pascal library. We recommend that you display or print the *GEMSUBS.PAS* file to become familiar with its contents.

The PASGEM library file is a standard TOS object library, and you can use either the Personal Pascal linker or the Digital Research L68 linker from Atari to link your object files with the GEM/Pascal library or any other libraries you may need.
**SEE:** *The Linker* page 5-1.

**NOTE:** You can place the GEM/Pascal library files on any disk drive, in any directory or subdirectory. Just be sure to specify the full path name when you include or link with them.

**For example:**

```
{$I C:PASCAL\LIBRARY\GEMCONST}
```

## INITIALIZING AND EXITING GEM

A Personal Pascal program using any of the GEM procedures or functions described in this section must tell GEM when it is active and when it is finished.

## INIT_GEM

FUNCTION Init_Gem : integer ;

i := Init_Gem

Init_Gem requires no parameters. It tells GEM your program wants to work within the GEM environment. Init_Gem opens a virtual workstation to the screen, it performs the AES call APPL_INIT, returning the AES application ID. This ID is of interest only with desk accessories. If Init_Gem is successful, your program can make GEM calls.

If Init_Gem can't initialize your application, it returns a negative one (-1) as an error flag, and your only alternative may be to abort all operations.
**SEE:** The program example, page 7-4.

## EXIT_GEM

PROCEDURE Exit_Gem ;

This procedure is the last operation before exiting GEM. Exit_Gem assures that GEM knows you will be making no more requests of it.

**NOTE:** Do not call Exit_Gem if the call to *Init_Gem* was unsuccessful.
**SEE:** The program example, page 7-4.

# ALERT BOXES

The easiest GEM facilities to use are the alert boxes. Because of for their simplicity, alert boxes are limited in what they can do. They are best used to issue errors or warnings, to make simple two-way or three-way decisions, or to pause a program until the user is ready to continue.

## DO_ALERT ()

```
FUNCTION Do_Alert ( Prompt : STRING ;
                    Default: Short_Integer )
                    : Short_Integer ;
```

**EXAMPLE:**

```
Choice := Do_Alert
          ('[1][What color?][ Red | Green | Blue ]', 0);
```

Do_Alert requires only two parameters, a string and an integer. The string is broken into three parts. Each part of the string passed to Do_Alert must be enclosed in square brackets.

The first part must contain only one ASCII character, a '0', '1', '2', or '3'. This character must be a literal part of the string and may NOT be represented by a variable name or other expression. This character tells GEM which of three possible icons to display within the alert box.

**ALERT ICONS**

| Character | Icon displayed |
|---|---|
| 0 | No icon displayed |
| 1 | An exclamation point in a diamond. |
| 2 | A question mark in an inverted triangle. |
| 3 | The word 'STOP' in a stop sign. |

*DO_ALERT ( Continued )*

The second part of the string is the prompt message to be displayed. Up to 5 lines of text may be displayed. GEM uses the vertical bar | as a separator between the prompt lines. The text in the prompt may not include either the vertical bar or a right square bracket.

```
[Look:|  More than|  one line!]
```

will cause this prompt within an alert box:

```
Look:
  More than
  one line!
```

The third part of the string consists of one, two, or three button names separated by vertical bars. The example, above, contains three button names: *Red*, *Green*, and *Blue*. The alert box will contain three evenly spaced rectangles containing the button names in order.

**NOTE:** Putting a space on each side of a button's name improves its appearance.

**ALSO:** Using a string variable for the prompt makes an alert box more versatile. Be careful about string length, however.

*DO_ALERT ( Continued )*

The integer parameter selects a button as the default choice. *Do_Alert* typically returns an integer value showing which button was clicked. If there is a value of 1, 2, or 3 in the integer parameter, the user can select that option by pressing the Return key.

**CAUTIONS:**

1 The total length of the string may not exceed 255 characters.

2 No text line in the prompt may exceed 40 characters, even fewer if the box contains an icon, and there may be no more than 5 lines.

3 The total length of all the button names may not exceed 20 characters.

4 Selection values may not be outside the expected range of 0 through 3.

5 If your program can run in low resolution, limit your text lines to 20 characters and your buttons to 10 characters. ( No alert box may exceed 25% of the total screen size.)

Violating any of these restrictions can cause unpredictable results, including a system crash.

# DIALOG BOXES

Dialog boxes are used to communicate with the user. They may contain user response items, or simply give information. To build a dialog box, or a message box, you must observe the order described in the following pages.

## BUILDING A DIALOG BOX

1 Reserve space for a dialog tree descriptor using *New_Dialog*. This gives you a descriptor pointer to use with subsequent GEM routines.

2 Describe the general items to appear in the box, using *Add_DItem*.

3 Describe text fields, using *Set_DText* for fixed text, and *Set_DEdit* for editable text. Use *Obj_SetState* to refine the appearance and capabilities of the box.

4 Draw a dialog box on the screen by calling *Do_Dialog*, and wait for the user to respond. Use Show_Dialogl if you are not going to pause for the user to respond. To center the box on the screen, use *Center_Dialog* before using *Do_Dialog* or *Show_Dialog*.

5 If a dialog box contains an editable text field, use *Get_DEdit* to get the user's input. Use *Obj_State* to determine the state of other items.

6 Modify items in a box by using *Obj_SetState* or *Obj_SetFlags*. Reactivate the the box with *Redo_Dialog*, then go to step 5.

*BUILDING A BOX ( Continued )*

7 When you're done with the box, erase it from the screen using *End_Dialog*.

8 If you won't be using a box again, or if you need to restore editable fields to their original form, use *Delete_Dialog* to release the space used by the dialog or message box descriptor. You can routinely perform this step, since starting again from step 1 will produce the same box.

**NOTE:** If a routine stores a dialog tree pointer in a local variable, you *SHOULD* release the space when it is done. Otherwise, it becomes unavailable memory space.

## NEW_DIALOG ()

```
i := New_Dialog ()

FUNCTION New_Dialog (item_count,
                     left, top,
                     width, height : Short_Integer)
                     : Dialog_Ptr ;
```

**EXAMPLE:**

```
name_box := New_Dialog ( 8, 0, 0, 40, 10 ) ;
```

New_Dialog returns a pointer to memory reserved for a box descriptor. If it can't allocate the needed space, it returns zero. New_Dialog requires 5 integer parameters:

1 The number of items to place in the box. Choose a number a little higher than you expect to use so you can add items without having to remember to change the New_Dialog parameters.

2 The horizontal position of the top left corner of the box relative to the upper left corner of the screen. The position is given in characters, not pixels.

*NEW_DIALOG ( Continued )*

3 The vertical position of the top left corner of the box. The position is given in characters, not pixels.

4 The width of the box, in characters.

5 The height of the box, in character lines.

**NOTE:** If you intend to center the dialog box, you need not calculate the corner coordinates. Use 0,0 with *New_Dialog* and call *Center_Dialog* to cause automatic centering.

## ADD_DITEM ()

```
i := Add_DItem ()

FUNCTION Add_DItem ( box : Dialog_Ptr ;
                     item_type,
                     flags,
                     item_left, item_top,
                     item_width, item_height,
                     border,
                     color : Short_Integer )
                     : Short_Integer ;
```

**EXAMPLE:**

```
Red_Button := Add_DItem ( Color_Box,
                          G_Button,
                          Radio_Btn | Selectable,
                          4,4,
                          3,1,
                         1,
                          my_palette ) ;
```

Use Add_DItem to add an item to a dialog box. Space for the box descriptor must have been reserved by *New_Dialog.* Add_Ditem requires 9 parameters.

Add_DItem returns an integer value which is the new item's displacement within a dialog tree. When associated with the dialog pointer, it provides a unique item identifier for subsequent GEM calls.

*ADD_DITEM ( Continued )*

Because there are so many options available to *Add_DItem*, we give only a brief summary of each parameter here. Options requiring further explanation are explained further in the following pages.

*Add_DItem* parameters are:

**1** A pointer to a dialog box tree, previously obtained by *New_Dialog*.

**2** An object type number.
**SEE:** *Object Types* for *Add_DItem*, below.

**3** An object characteristic flag block.
**SEE:** *Object Flags* for *Add_DItem*, below.

**4** The horizontal location of the top left corner of the item, in character coordinates, relative to the top left corner of the dialog box.

**5** The vertical location of the top left corner of the item, in character coordinates, relative to the top left corner of the dialog box.

**6** The width of the item, in characters. Displayed text may not exceed this limit. Excess text will be clipped.
**SEE:** *G_String*, page 7-21 for an exception.

*ADD_DITEM ( Continued )*

**7** The height of the item, in character lines.

**8** The width of the border around the item. Zero specifies no border. A positive value, from 1 to 127, thickens the border inward. A negative value, from -1 to -127, thickens the border outward.

**NOTE:** Practical border widths on an Atari ST computer are -4 through +4; the heavy border around the default button in an alert box has a width of 3.

**9** The item color block. Used to specify item color, background color, border color, and more.
**See:** *D_Color*, page 7-24.

**NOTE:** Boxed text looks good with a box height of 1 if the width of the border is negative. For a height of 2, use a positive border width. Boxed text with a height of one will not be drawn properly when the border has a positive width. Also; editable boxed text does not look good with a height of one since the text cursor's height is slightly greater than one.

# blank page

## OBJECT TYPES FOR ADD_DITEM

These object types are mutually exclusive. Give the name of one, and only one of them, as the second parameter to *Add_DItem.*

**G_IBox** The outline of a box. The box will be of the size and border color passed by the other parameters. It will be placed at the specified location within the dialog box. If the border width is zero, the box will be invisible.

**G_Box** Same as *G_IBox*, but the box is filled with the fill color and pattern specified.

**G_Text** A line of text to be specified by a subsequent call to *Set_DText.* Left, center, and right justified text are all available.
**SEE:** *Set_DText.* 7-26

**G_BoxText** Same as *G_Text* except that the text will have a box of the specified border color drawn around it. It can also have a fill color and pattern, as can *G_Box.*

**G_FText** A line of editable text to be specified by a subsequent call to *Set_DEdit.* Left, center, and right justified text are all available.
**SEE:** *Set_DEdit* page 7-28.

*OBJECT TYPES FOR ADD_DITEM ( Continued )*

**G_FBoxText** Same as *G_FText* except that the text will have a box drawn around it in the border color specified. You can specify a fill color and pattern, as with *G_Box_Text.*

**G_String** A simplified form of *G_Text.* Border size, item width, and color are ignored and the text is always left justified. The text must be given in a subsequent call to *Set_DText.* The *Set_DText* string size determines the item width.

**G_Button** A simplified form of *G_BoxText.* Border width and color parameters are ignored. The border width is assumed to be 1 unless the button is selected, in which case it is assumed to be 3. The text within the button must be given in a subsequent call to *Set_DText* and is always centered.

## OBJECT FLAGS FOR ADD_DITEM

The items in a dialog box usually have flags associated with them to define their characteristics. Object flags are individual bits within a word, and selecting more than one flag implies *OR*-ing the flags together, as with the flags *Radio_Btn | Selectable*, in the previous example. If you frequently use a particular combination of flags, you may want to construct a new CONST describing it, so the hexadecimal values of the various flags are given here:

**None** ($00) For program readability only, since zero is the lack of any flags.

**Selectable** ($01) If you make an item selectable, GEM highlights it when the user clicks on it, by reversing its foreground and background colors.

**NOTE:** Several of the following flags require this flag to be set for them to be meaningful.

**Default** ($02) Only one item in a box may have this flag set. That item will be selected if the user presses the Return key. If a button is assigned this flag, its border width is set to 3.

*OBJECT FLAGS FOR ADD_DITEM ( Continued )*

**Exit_Btn** ($04) If a selectable item is flagged as an Exit_Btn, Gem will highlight it and exit the dialog when the user clicks on it.

**Editable** ($08) An editable text field. This flag is valid only for *G_FText* or *G_FBoxText* item types.

**Radio_Btn** ($10) When two or more buttons in the dialog box are given this flag, only one of them may be selected at a time. When the user clicks on one radio button, it is selected and all other radio buttons are de-selected.

**NOTE:** Radio buttons must also be Selectable.

**Touch_Exit** ($40) Similar to *Exit_Btn* except that the user need only move the mouse pointer to the object and press the mouse button. GEM does not wait for the button to be released.

GEM uses the other bits of the flag word for its own purposes. Improper flag use can cause a system crash.

## D_COLOR ()

```
i := D_Color ()

FUNCTION D_Color ( border,
                   text: Short_Integer;
                   text_mode: Boolean;
                   pattern,
                   fill: Short_Integer )
                   : Short_Integer
```

**EXAMPLE:**

```
my_palette := D_Color ( Black, Red, 1, 5, White );
```

D_Color computes the object color value to be used with *Add_DItem*. This relieves you of the task of computing, shifting and *OR*-ing color values yourself. Here is a list of the predefined colors and their values:

| | | | |
|---|---|---|---|
| **White** | **0** | **L_White** | **9** |
| **Black** | **1** | **L_Black** | **10** |
| **Red** | **2** | **L_Red** | **11** |
| **Green** | **3** | **L_Green** | **12** |
| **Blue** | **4** | **L_Blue** | **13** |
| **Cyan** | **5** | **L_Cyan** | **14** |
| **Yellow** | **7** | **L_Yellow** | **15** |
| **Magenta** | **8** | **L_Magena** | **16** |

The L_ prefix indicates a light shade of a color.

*D_COLOR ( Continued )*

**NOTE:** If the user has altered the default color values, or if your program has previously used Set_Color to change them, the names in the color list may bear no real relationship to the color names.

The PATTERN parameter sets the intensity of the fill color. A value of 0 is no color at all, while 7 selects a solid color. The expected value range is 0 to 7.

The Boolean parameter, *text_mode* is 0 for transparent mode, and 1 for replace mode. Transparent mode combines text with background, leaving both visible. Replace mode over-writes any background color or pattern. You should use transparent mode with fill pattern values other than 0.

**NOTE:** Some object types, such as *G_String*, ignore some or all of the information that *D_Color* provides.

## SET_DTEXT ()

```
PROCEDURE  Set_DText ( box : Dialog_Ptr;
                       item : TreeIndex;
                       text : Str255;
                       font : Short_Integer;
                       justify : TE_Just )
```

**EXAMPLE:**

```
Set_DText ( instruct_box, line1, 'Instructions:',
                Small_Font, TE_Center ) ;
```

Set_DText puts the specified text into the proper item in a box. The first parameter must be a dialog pointer obtained by *New_Dialog*. The second parameter must be the index of an item within the box, previously obtained by *Add_DItem.*

The fourth parameter specifies the font size and type to use for the text. The only legal values are *System_Font* (3) and *Small_Font* (5). Using other values can produce disastrous results.

The last parameter specifies the text justification within the boxed area. There are only three legal values:

| | |
|---|---|
| **TE_Left** | specifies left justify |
| **TE_Right** | specifies right justify |
| **TE_Center** | specifies center justify (centered) |

*SET_DTEXT ( Continued )*

**NOTE:** Some object types, such as *G_String* and *G_Button* ignore the justification parameter and always use *System_Font*. You must still code these parameters, even though they are ignored.

**NOTE ALSO:** You may change the text in any item by making calls to *Set_DText*, but you must be sure to limit the new string length to the original string length. If the new string is longer than the original one, a system crash may result.

## SET_DEDIT ()

Set_DEdit ()

```
PROCEDURE  Set_DEdit( box : Dialog_Ptr;
                      item : Tree_Index;
                      template : Str255;
                      validation : Str255;
                      initial  : Str255;
                      font : Short_Integer;
                      justify : TE_Just )
```

**EXAMPLE:**

```
Set_DEdit ( name_box, line1,
            'Date: __/__/__',
            '999999',
            '102685',
             System_Font,
             TE_Left ) ;
```

Set_DEdit describes an editable text field in a dialog box. The first two parameters must be a valid dialog tree pointer and item index. Set_DEdit also requires three text strings:

The first string parameter specifies the *input template* that appears in the dialog box. All characters except the underbar (_) appear exactly as they will on the screen. The underbar characters indicate the places that the user can enter text. In the example, there are six characters of editable text available to accept input.

*SET_DEDIT ( Continued )*

The second string parameter is the *validation string.* Each character of this string specifies the type of character that GEM will accept in the corresponding position on the screen. There must be exactly as many characters in the validation string as there are underbar characters in the template string.

The validation characters restrict user input as shown:

| Character | Meaning |
|---|---|
| **9** | Accept only decimal numeric digits. |
| **A** | Accept only upper case alphabetic characters or a space |
| **a** | Accept only upper case and lower case alphabetic characters or a space |
| **N** | Accept digits, upper case alphabetic characters, or a space. |
| **n** | Accept digits, upper case and lower case alphabetic characters, or a space. |
| **F** | Accept any character that can be part of a GEMDOS FILE name. |
| **P** | Accept any character that can be part of a GEMDOS PATH name. |
| **p** | Same as P, but do not accept wild card characters, '*' and '?'. |
| **X** | Accept any printable character. |

*SET_DEDIT ( Continued )*

The third string parameter defines the *initial* set of characters to display. For example, you may want to have the user specify a date. If you place today's date in the initial string, the user need only press Return to select it. There should be no more characters in this string than there are underbar characters in the template string, but there may be fewer. You can even specify a null string. If there are fewer characters in this string than there are underbars in the template string, the extra underbars will be displayed.

The next parameter specifies the *font size* to use for the text. Again, the only legal values are *System_Font* (3) and *Small_Font* (5). Using other values can produce disastrous results.

*SET_DEDIT ( Continued )*

The last parameter specifies text justification. There are only three legal values:

| | |
|---|---|
| **TE_Left** | specifies left justify |
| **TE_Right** | specifies right justify |
| **TE_Center** | specifies centered text |

**NOTE:** You may call *Set_DEdit* only once per item. Any calls after the first one for a given item will be ignored. Use *Set_DText* to change the initial string.
**SEE:** *Obj_ReDraw* page 7-40.

**CAUTION:** Some versions of Atari GEM will crash if you program the *F*, *N*, or *n* as validation characters and the user types an *underline* character as part of the response to the dialog. It's a bit painful, but *much* safer, to use the *X* parameter and perform your own validation.

## OBJ_SETSTATE ()

```
PROCEDURE Obj_SetState ( box : Dialog_Ptr;
                         item : Tree_Index;
                         state : Short_Integer;
                         redraw : Boolean ) ;
```

**EXAMPLE:**

```
Obj_SetState ( option_box, button3,
                Outlined | Shadowed, False ) ;
```

Besides specifying object type and object flags with *Add_DItem*, you can also specify the *state* of an item in a dialog box using Obj_Set_State. The first two parameters are a valid *dialog tree pointer* and a valid *item index* obtained by *New_Dialog* and *Add_DItem*. The third parameter specifies an object's *state* and may be of one of the following values or, as in the example, a valid *OR*-ed combination of them:

**Normal** ($00) None of the following

**Selected** ($01) Background and foreground reversed.

**Crossed** ($02) Draw an "X" from corner to corner in the specified box.

**CAUTION:** Some versions of GEM have problems with this option.

*OBJ_SETSTATE ( Continued )*

**Checked** ($04) Draw a check mark in the box represented by this item. The check is drawn in the center of a box if the box contains no text. If the item contains text, the check is drawn in the first character position of the text field. It's a good idea to start text with two space characters, if it might be checked later.

**Disabled** ($08) Draw any text in half intensity, giving the appearance of ghost text.

**Outlined** ($10) Outline an entire box. This sets it off more from the surrounding fields. Dialog boxes are drawn in this state.

**Shadowed** ($20) If the item includes a box, the right and bottom sides of the box are outlined. Again, this sets it off more from the surrounding fields.

**NOTE:** Many of these states do not make sense with some item types. Use them with care.

The last parameter to this routine is a Boolean flag, *True* or *False*, which indicates whether GEM needs to redraw the object. Do not ask for a redraw of an item in a dialog box which has not yet been drawn! Only ask for a redraw after you have called *Do_Dialog* and before you make a call to *Redo_Dialog*.

## CENTER_DIALOG ()

PROCEDURE Center_Dialog ( box : Dialog_Ptr ) ;

**EXAMPLE:**

```
Center_Dialog ( name_box );
```

Center_Dialog causes GEM to center the specified box on the screen. This procedure avoids the need to calculate where the box will fit on a given screen, monochrome, low-resolution color, or high-resolution color. You need not give coordinates for the upper left corner position of the specified dialog box.
**SEE:** *New_Dialog* page 7-14.

## DO_DIALOG ()

ptr := Do_Dialog ()

```
FUNCTION Do_Dialog ( box : Dialog_Ptr;
                     start_item : Tree_Index )
                     : Tree_Index ;
```

**EXAMPLE:**
pushed := Do_Dialog ( color_box, red_button ) ;

To display a dialog box, simply call Do_Dialog.

The first parameter must be a dialog box pointer previously obtained by *New_Dialog*.

The second parameter specifies the item number of an editable text field defined by *Add_DItem*. The cursor will be put in that field when the dialog is executed.If you don't want the cursor in the field when the dialog box first appears, or there is no editable field in the box, you must use 0 for this parameter.

**CAUTION:** Invalid values cause Atari GEM to crash.

Do_Dialog returns the index of the item causing a return. If the user edited a text field and then pushed Return, Do_Dialog would return the item number of the field. If the user clicked an exit button, Do_Dialog would return the item number of the button.
**SEE:** *Add_DItem* and its Object States, page 7-16.

## SHOW_DIALOG ()

PROCEDURE Show_Dialog ( box : Dialog_Ptr; )

**EXAMPLE:**
Show_Dialog ( info_box ) ;

To display a dialog box without pausing your program for user input, use Show_Dialog. The parameter must be a box pointer previously obtained by *New_Dialog*.

**NOTE:** When you use *Show_Dialog*, editable fields are displayed and object states are displayed, but the fields can't be edited and the object flags are meaningless.

## GET_DEDIT ()

```
PROCEDURE Get_DEdit ( box : Dialog_Ptr ;
                      item : Tree_Index ;
                      VAR edited : Str255 ) ;
```

EXAMPLE:

```
Get_DEdit ( name_box, line1, edited_name ) ;
```

Get_DEdit finds out the current contents of an editable field in a dialog box. It requires a dialog box tree pointer and an item offset previously obtained by *New_Dialog* and *Add_DItem*. The third parameter must be a variable of the appropriate string type to receive the edited text field.

## OBJ_STATE ()

```
i := Obj_State ()

FUNCTION  Obj_State ( box : Dialog_Ptr ;
                      item : Tree_Index )
                      : Short_Integer ;
```

**EXAMPLE:**

```
IF Obj_State ( op_box, button3 )
                & Selected <> 0 THEN ...;
```

When *Do_Dialog* or *Redo_Dialog* return, they return the identifier of the exit item that terminated the dialog. They don't tell you the state of any other items in the box. To find the state of any item in the dialog box, pass Obj_State the dialog pointer and item number previously obtained by *New_Dialog* and *Add_DItem*. The state of that item will be returned as an integer.

The bits within the returned integer have the same meanings described in the discussion of *Obj_SetState*. You can use the Personal Pascal bit manipulation operators to determine whether a particular state is true, as in the example.

**NOTE:** If a selectable item can cause an exit from a dialog, you must use Obj_State to test for it and de-select it before calling *Redo_Dialog*. Otherwise, the user will have to click the object twice, first de-selecting it and then re-selecting, before an exit will occur. This note also applies if the dialog is terminated by *End_Dialog* and later reinstated by another call to *Do_Dialog*.

**ALSO:** If you intend to change the state of an item after using *Do_Dialog*, and before a call to *Redo_Dialog*, you must call *Obj_SetState* with the redraw parameter set True, or the item will not be redrawn in its new state. Forgetting to do that is not inherently dangerous, but the screen display will not reflect GEM's state and will probably confuse the user.

## OBJ_REDRAW

```
PROCEDURE Obj_ReDraw( box : Dialog_Ptr ;
                      item : Tree_Index ) ;
```

All dialog box items except text fields may be modified (via *Obj_SetFlags* and *Obj_SetState*) and then "redrawn" (i.e., updated on the screen) by coding the last parameter of *Obj_SetState* as *TRUE.*

Text items ( those created with *Set_DText* or *Set_DEdit* ) may be modified ( via a subsequent call to *Set_DText* ), but they are NOT automatically redrawn (even by a call to *Redo_Dialog* ). Obj_ReDraw provides the means to redraw these items. In fact, Obj_ReDraw may be used to redraw ANY item (thus avoiding the complication of a call to *Obj_SetState* ).

Only two parameters are required: The dialog box pointer and the item index for the item to be redrawn.

## OBJ_FLAGS ()

i := Obj_Flags

FUNCTION Obj_Flags ( box : Dialog_Ptr ;
item : Tree_Index )
: Short_Integer ;

**EXAMPLE:**

curflags_3 := Obj_Flags ( curbox, item_3 ) ;

To find the state of the flags for any item in a dialog box, pass Obj_Flags the dialog pointer and item number previously obtained by *New_Dialog* and *Add_DItem.* Obj_Flags will return the flag integer for that item.

**NOTE:** This function may seem to be of little use, since you specify the flags and the user can't alter them, but if you load a dialog box from a resource file, Obj_Flags provides the best way to ensure that the program accurately reflects the state of the dialog box. It is also a convenient way to obtain flag bits without having to establish and maintain global variables.
**SEE:** The example in *Obj_Setflags* page 7-42.

## OBJ_SETFLAGS ()

```
PROCEDURE Obj_SetFlags ( box : Dialog_Ptr ;
                         item : Tree_Index ;
                         flags : Short_Integer ) ;
```

**EXAMPLE:**

```
Obj_SetFlags (curbox,
              item_3,
              Obj_Flags (curbox, item_3) | Default ) ;
```

Obj_SetFlags changes the state of the flags associated with an item. It's generally used after *Do_Dialog* or *Redo_Dialog* to reflect changes in the state of a dialog box in response to user actions.

The first two parameters are a dialog pointer and an item number previously obtained by *New_Dialog* and *Add_DItem.*

The third parameter is an integer whose bits have the meanings described in the section on *Object Flags.*
**SEE:** *Add_DItem* page 7-16.

**CAUTION:** Since GEM maintains some of the bits in the flag integer, you should not simply assign a new value to a given flag word. Instead, use Personal Pascal's bit manipulation operators to turn individual bits on or off, as in the example.

## REDO_DIALOG ()

```
i := Redo_Dialog ()

FUNCTION  Redo_Dialog ( box : Dialog_Ptr,
                        start_item : Tree_Index )
                        : Tree_Index ;
```

**EXAMPLE:**

```
pushed := Redo_Dialog ( color_box, red_button ) ;
```

Use Redo_Dialog exactly as you use *Do_Dialog*. The difference is that Redo_Dialog will NOT cause the dialog box to be displayed again. Use this routine only *after* a call to *Do_Dialog* and *before* a call to *End_Dialog* clears the dialog box from the screen.

**NOTE:** Although you can change the text for any given item using *Set_Dtext*, you can *NOT* change an editable text field.

**ALSO:** You *must* call *Obj_SetState* to redraw any objects you have changed.
**SEE:** the sections on *Obj_SetState* and *Obj_State* for special notes and cautions about the states of dialog items before using Redo_Dialog.

**FURTHER:** You must call *Obj_ReDraw* to redraw any text fields you have changed.
**SEE:** *Obj_ReDraw*, page 7-40.

## END_DIALOG ()

PROCEDURE End_Dialog ( box : Dialog_Ptr ) ;

**EXAMPLE:**

End_Dialog ( name_box );

This procedure removes a dialog box previously placed on the screen by *Do_Dialog.* or *Show_Dialog.*

**NOTE:** Erasing a dialog box will cause underlying windows to reappear. Unfortunately, GEM doesn't redraw them automatically. Instead, GEM tells YOUR program to redraw the window.
**SEE:** *Get_Event*, page 7-150.

**Remember:** To reuse a closed box, call *Do_Dialog* or *Show_Dialog* again. It will be restored just as it was.

**CAUTION:** Editable fields will be restored as the user left them. Dialog boxes containing editable fields are usually deleted and rebuilt rather than reinstated by *Do_Dialog.* You might also need to reset object states with *Obj_SetState* before the box is re-established.

## DELETE_DIALOG ()

PROCEDURE Delete_Dialog ( box : Dialog_Ptr ) ;

**EXAMPLE:**

Delete_Dialog ( name_box );

Delete_Dialog releases the space reserved by *New_Dialog*. Delete_Dialog is intelligent: it also releases *all* space used by items in the box created by *Add_DItem, Set_DText*, etc.

Once a dialog box has been deleted by this procedure, you must re-execute *all* the code which built it, starting with *New_Dialog*, to reinstall it.
**SEE:** The caution about editable fields in the *End_Dialog* description, page 7-44.

# PREDEFINED DIALOG BOXES

Many programs need to ask the user for file names. To promote consistency in programs written using Personal Pascal, the GEM/Pascal library implements two ready-to-use dialog boxes.

## GET_IN_FILE ()

```
b := Get_In_File ()

FUNCTION  Get_In_File ( VAR path : Path_Name ;
                        VAR name : Path_Name )
                        : Boolean ;
```

**EXAMPLE:**

```
IF Get_In_File ( def_path, full_name ) THEN ... ;
```

GEM provides a standard item selector which draws a *dialog box* and asks the user for the name of an existing file. The user may type in a file name, use the mouse to select the file without typing its name, change directories, drives, etc.

The GEM/Pascal library routine Get_In_File makes this dialog box accessible to the Personal Pascal programmer. Provide Get_In_File with a default path name and the dialog box will automatically display files in the subdirectory at the end of that path. The user can change the path name at will.

When Get_In_File returns, the file name is contained in the second variable parameter. The path specifying the subdirectory containing that name is returned as the first variable parameter. If the user canceled the dialog, the function returns a *False* value; otherwise, it returns *True*.

## GET_OUT_FILE ()

```
b := Get_Out_File ()

FUNCTION  Get_Out_File ( prompt : String ;
                         VAR name : Path_Name )
                         : Boolean ;
```

**EXAMPLE:**

```
IF Get_Out_File ( 'Write to ...', out_name ) THEN ...
```

Get_Out_File doesn't encourage the user to choose an existing file name, so its dialog box is simpler. The program can supply a string as a prompt. The user may still specify a path name and change disks in the process of producing a file name.

If the file name entered by the user already exists, Get_Out_File opens an alert box telling the user so and asks for verification of the file name. If the user chooses not to re-use the existing name, Get_Out_File provides another chance to enter one.

If the user clicks the cancel button, Get_Out_File returns a *False* value; otherwise, it returns *True.*
**SEE:** *Get_Out_File* in the Appendix.

# MENUS

## INTRODUCTION TO THE MENU BAR

To establish and maintain a menu bar:

Allocate space for a menu tree, give it a name, and get its pointer with *New_Menu*. Page 7-52.

Place titles in the menu bar using *Add_MTitle*. page 7-53.

Add items to the menus with *Add_MItem*. Page 7-55.

Highlight, enable, or check-mark appropriate menu items using *Menu_Hilight, Menu_Enable*, and *Menu_Check*. Pages 7-60, 7-58, 7-57.

Use *Draw_Menu* to display the menu bar at the top of the screen. Page 7-62.

While the menu is active, your program can modify any of the options selected in step 4. You can also change the text of any menu item as necessary to reflect program's state.

When you're done with the menu, clear it from the screen using *Erase_Menu*. Page 7-64.

If appropriate, release the memory space using *Delete_Menu*. Page 7-65.

## MENUS ( Continued )

Putting a menu on the screen is easy. Using it is harder. Menu bar options are usually activated by a mouse event reported by *Get_Event*. To use menu bars, you must understand event management. However, when you read about event handling, knowing how to set up and enable a menu bar will make the process clearer, so read this part first, then read the section on Event Management.

**NOTE:** A GEM program need not have a menu bar! One advantage of a GEM program without a menu bar is that the user can't load desk accessories. That may avoid having to redraw parts of the screen. The advantages of menus usually outweigh this consideration for all but the most graphics-oriented programs.

BLANK
PAGE

## NEW_MENU ()

```
p := New_Menu ()

FUNCTION  New_Menu ( count : Short_Integer ;
                     about : Str255 )
                     : Menu_Ptr ;
```

**EXAMPLE:**

```
a_menu := New_Menu ( 30, ' About my program ' ) ;
```

New_Menu only reserves memory for the GEM menu tree. The integer parameter specifies the maximum number of titles and items combined that will appear in the menu. It's a good idea to make this number a bit larger than seems necessary, so you can add the inevitable one or two items without running out of room.

The second parameter is the string which will be installed on the first line of the drop-down menu when the *Desk* title is selected.

The value returned by New_Menu is a menu tree pointer to be used in all further calls relating to the menu.

## ADD_MTITLE ()

```
i := Add_MTitle ()

FUNCTION  Add_MTitle ( menu : Menu_Ptr ;
                       title : Str255 )
                       : Short_Integer
```

**EXAMPLE:**

```
title1 := Add_MTitle ( a_menu, ' Files ' ) ;
```

Add_MTitle establishes a title in the menu bar. As you add titles to your menu bar, be sure the total number of characters and separating spaces in all the titles does not exceed the screen size. For flexibility, limit the total to 40 characters, including *Desk*, since that's the maximum that fits a low resolution color display.

Add_MTitle requires two parameters; a menu pointer, previously obtained by *New_Menu*, and a *string* containing be the *title.* The integer value returned is an index to this title. This integer must be used with every subsequent menu routine referring to this title.

**NOTE:** The *Desk* title on the left side of the menu bar is put there by GEM and can't be changed.

**ALSO**: If you put a space at each end of a title, it will serve to separate the titles on the menu bar and allow the drop-down menu to be presentably indented.
**SEE:** *Add_MItem*, page 7-55.

**CAUTION:** *All* menu titles *must* be added by calls to *Add_MTitle* before *any* items are added with *Add_MItem.*

## ADD_MITEM ()

```
i := Add_MItem ()

FUNCTION  Add_MItem ( menu : Menu_Ptr ;
                      title_index : Short_Integer ;
                      item : Str255 )
                      : Short_Integer ;
```

**EXAMPLE:**

```
save_item := Add_MItem ( a_menu, title1,
                         ' Save as... ' ) ;
```

Add_MItem adds an item to the drop-down menu associated with a menu title. The total number of items in a menu may not exceed the screen display area. This allows up to 24 items in a menu, but presenting more than six or eight items in a menu becomes unwieldy and confusing to the user.

**NOTE:** The total space occupied by a single drop-down menu may not exceed *one quarter* (25%) of the screen space.

Add_MItem requires 3 parameters: a menu tree pointer, previously obtained by *New_Menu,* a menu title index returned by *Add_MTitle*, and a *string* containing the *item name.* The value returned is an integer index to this item. This integer must be used in any call refering to this item, including event handling routines testing to see if this item was selected.

NOTE: For esthetic purposes, item names should be preceded by two spaces to indent them when they drop down. The longest item name should have one or two spaces following it and all other items should be padded with blanks to that same length. Otherwise, when GEM highlights a short item, the highlighting won't go the full width of the column and it will look maximally ugly. It's also a GEM convention to use two spaces to allow for possible check-marks.
**SEE:** *Menu_Check* page 7-57.

**CAUTION:** Items must be added in groups, by parent title, *in the same order that the titles were added.* There is a bug in GEM that causes strange and displeasing results if you break this rule.

## MENU_CHECK ()

```
PROCEDURE Menu_Check ( menu : Menu_Ptr ;
                       item_index : Short_Integer ;
                       check : Boolean )
```

**EXAMPLE:**

```
Menu_Check ( a_menu, protect_item, True ) ;
```

Menu_Check adds or removes a check mark character preceding any item in the menu. The first two parameters specify the menu pointer and index for the item to be checked. The third parameter must be *True*, to enable the check mark, or *False*, to disable it. It is not an error to enable a check which already exists or to disable one which does not exist.

The check mark is drawn at the left side of the item, so the titles of items which may later be checked should begin with two spaces.

**SEE:** *Add_MItem,* page 7-55.

**MENU_ENABLE ()**
**MENU_DISABLE ()**

```
PROCEDURE Menu_Enable ( menu : Menu_Ptr ;
                        item_index : Short_Integer )

PROCEDURE Menu_Disable ( menu : Menu_Ptr ;
                         item_index : Short_Integer )
```

**EXAMPLES:**

```
Menu_Enable( a_menu, saveas_item ) ;
Menu_Disable( a_menu, readfrom_item) ;
```

To enable or enable a menu item, use the appropriate routine and pass the appropriate menu bar pointer and item index, previously obtained by *New_Menu* and *Add_MItem.*

A menu item which has been *disabled* is displayed in half intensity. *Enabled* items are displayed full intensity. Disable menu items which your program is to consider unavailable. For example, an editor might disable Save as... until there is text in memory.

**NOTE:** These routines may be used either before or after the menu bar is actually drawn on the screen.

**ALSO:** The horizontal bar that you often see in drop-down menus is not a special kind of item. It's just a line of hyphens placed in the menu by *Add_MItem* and then disabled by *Menu_Disable.* You can produce other interesting effects by disabling lines of other characters.

**MENU_HILIGHT ()**
**MENU_NORMAL ()**

```
PROCEDURE Menu_Hilight ( menu : Menu_Ptr  ;
                         title_index : Short_Integer )

PROCEDURE Menu_Normal ( menu : Menu_Ptr ;
                        title_index : Short_Integer )
```

**EXAMPLES:**

```
Menu_Hilight ( a_menu, files_title ) ;
Menu_Normal ( a_menu, options_title ) ;
```

Use these routines to highlight a menu title or restore it to normal. Pass the menu pointer and item index previously obtained by *New_Menu* and *Add_MTitle.*

A *highlighted* menu title is displayed in inverse video. *Normal* titles are displayed in normal video.

**NOTE:** GEM usually highlights a menu title which has been selected by a mouse event. You can also highlight menu titles in response to key sequences. Since it is OK to highlight a title which has already been highlighted, it is fairly easy to write code to handle mouse and keyboard events similarly.

**ALSO:** The application program must return the menu title to normal mode by a call to *Menu_Normal.* GEM will not do it for you.

**FURTHER:** neither of these routines should be called until after *Draw_Menu* has activated the menu.

## DRAW_MENU ()

PROCEDURE Draw_Menu ( menu : Menu_Ptr )

**EXAMPLE:**

Draw_Menu ( a_menu ) ;

Draw_Menu displays the menu you've built.

Draw_Menu requires 1 parameter: a menu bar pointer.

The menu bar will remain visible until erased by *Erase_Menu.*

**CAUTION:** The menu bar *must* be erased before exiting your program, or the desktop will not be properly redrawn. If your program uses more than 1 menu bar, *always* erase the current one before displaying the next one.

## MENU_TEXT ()

```
PROCEDURE Menu_Text ( menu : Menu_Ptr ;
                      item_index : Short_Integer ;
                      new_text : Str255 )
```

**EXAMPLE:**

```
Menu_Text ( a_menu, verify_option, ' No verify ')
```

A menu item can contain text that needs to be changed to reflect the state of a program. Use Menu_Text to change the text.

Menu_Text requires 3 parameters: a *menu bar pointer* and *menu item index*, previously obtained by *New_Menu* and *Add_MItem*, and a *string* containing the new text.

**NOTE:** The new text string is actually copied over the old text, so it can't be longer than the old string. Also, if it is shorter, the remaining characters in the old text are not changed, so pad strings with spaces if necessary.

## ERASE_MENU ()

PROCEDURE Erase_Menu ( menu : Menu_Ptr )

**EXAMPLE:**

Erase_Menu ( a_menu ) ;

When you are through with particular menu bar, Erase_Menu will remove it from the screen. Erase_Menu requires one parameter, a *menu bar pointer.*

A menu bar remains visible until it is erased by Erase_Menu. It must be erased before GEM can properly draw another menu bar, including the desktop menu. That means you MUST erase your menu bar before exiting your program and returning to the desktop.

**NOTE:** The space reserved for a menu is not released until you call Delete_Menu, so you can have several menus set up and ready to swap as needed.

## DELETE_MENU ()

PROCEDURE Delete_Menu ( menu : Menu_Ptr )

**EXAMPLE:**

```
Delete_Menu ( a_menu ) ;
```

When you are through using menu, you can use Delete_Menu to release the space it used. Delete_Menu requires 1 parameter, a *menu pointer.* This procedure recovers the space used by the menu bar and all text strings used in the titles and items.

**NOTE:** There is generally no reason to use this routine. We recommend that you ignore it unless you have some special need to release unused menu space.

**CAUTION:** *Always* erase a menu before deleting it or grotesquely funny menus may result.

BLANK
PAGE

# WINDOW MANAGEMENT

This section discusses the routines that put windows on the screen. It doesn't cover routines to put graphics or text into a window.
**SEE:** *Window Text and Graphics*, page 7-97.

Some of the material in this section will become more clear when you read the section on *Event Management*, but having a rough knowledge of window management will help you understand events, so read this section first.

As with dialog boxes and menus, you must follow a particular sequence when using windows. The general sequence is given here, but it is important to remember that event occurrences can affect window management:

Use *New_Window* to reserve a window. p. 7-70

Use *Open_Window* to put the window on the screen. p. 7-76

Use *Set_Window* to select a window for graphics or text output. p. 7-82

Put information into the window using the calls described in the *Window Text and Graphics* section. Use *Set_Clip*, *Work_Rect*, *Border_Rect*, and *Set_WSize* to manage your window work areas. *Window Text and Graphics* is on p. 7-97.

If your window is overwritten by other windows, or by a dialog box, you must redraw the window. If your window is not the front one, this can be a very complex process, as described in the section on *Event Management*, page 7-133.
**SEE:** *Bring_To_Front; Front_Window* p. 7-92.

6. When you are temporarily through with a window, you can erase it from the screen using *Close_Window*. *Close_Window* only erases it, the window space is still reserved in memory. You can use *Open_Window* to reestablish the window.

7. If you are no longer going to use the window, use *Delete_Window* to release its memory space.

This is by no means a full explanation of windows. Be sure to read the following sections before trying to implement them.

**NOTE:** Throughout this and subsequent sections, reference is made to parameters which are labeled simply *x,y,w,h* and which are always integer values. Unless otherwise stated, these values specify the horizontal *(x)* and vertical *(y)* coordinates of the top left of a window together with the width *(w)* and height *(h)* of the window. These values are given in pixels. In a few cases which are clearly noted, these parameters may refer to a *portion* of a window instead of the entire window.

**ALSO:** The only time the *x,y* numbers are given as anything except absolute pixel coordinates relative to the top left corner of the screen, is when they are used in the various drawing routines presented in the next section. There, they are given relative to the top left corner of the window in which an object is drawn.

## NEW_WINDOW ()

```
I := New_Window ()

FUNCTION New_Window ( flags : Short_Integer ;
                      VAR title : Window_Title ;
                      x,y,
                      w,h : Short_Integer )
                      : Short_Integer ;
```

**EXAMPLES:**

```
big_window := New_Window ( G_All,
                           big_title,
                           0,0,0,0 ) ;

small_window := New_Window( G_Name | G_Close,
                            small_title,
                            300,100,100,50 ) ;
```

Use New_Window to create a new window, giving it a name, characteristics, and size and position constraints. New_Window doesn't draw the window on the screen. It tells GEM to reserve one of the available windows for its use. The procedure *Open_Window* actually draws the window.

**NOTE:** GEM has a maximum of 7 windows available at any time. This does not include alert boxes or dialog boxes.

## NEW_WINDOW PARAMETERS

The parameters to the New_Window function require extensive explanation:

The FLAGS variable is an integer whose bits control specific characteristics of the window being created. The name of each bit is declared as a constant in the GEM/Pascal library. The names and their characteristics are given on the following pages. The hexadecimal values of the names are also given so you can create your own flag names as hex CONSTants to combine particular sets of characteristics.

The descriptions on the following pages explain the window characteristics that appear if the corresponding flag bit is set to 1. If the bit is set to 0, the window will not include that characteristic.

**NOTE:** Use of the bitwise *OR* operator makes building combinations of characteristics easy.
**SEE**: the *small_window* example, page 7-70.

*NEW_WINDOW PARAMETERS ( Continued )*

**THE FLAG PARAMETER:**

**G_Name** ($001) A *title bar* will appear at the top of the window with the window name centered in it.

**G_Close** ($002) A *close box* will appear in the upper left corner of the window. This is the same close box that appears in the upper left corner of desktop directory windows.

**G_Full** ($004) A *full box* will appear in the upper right corner of the window. This is the same full box that appears in the upper right corner of desktop directory windows.

**G_Move** ($008) The user can move the window by clicking on the move box at the top of the window. This box is the one enclosing the title in the *title bar.*

**G_Info** ($010) A window may have a single line directly below the title bar in which miscellaneous information can be displayed. In the desktop, this line displays the *nnnn bytes used in nn files* information.

**G_Size** ($020) A *size box* will appear in the lower right corner of the window. This is the same size box which appears in the lower right corner of desktop directory windows.

*NEW_WINDOW PARAMETERS ( Continued )*

**THE FLAG PARAMETER:**

**G_UpArrow** ($040) Causes an upward-pointing arrow to appear on the right side of the window. Generally used for scrolling purposes, the up arrow appears in desktop directory windows.

**G_DnArrow** ($080) Causes a downward-pointing arrow to appear on the right side of the window. Generally used for scrolling purposes, the down arrow appears in desktop directory windows.

**G_VSlide** ($100) Causes the vertical slider on the right side of the window to appear. Usually used for scrolling purposes, the slider appears in desktop directory windows.

**G_LArrow** ($200) Causes the left pointing arrow on the bottom of the window to appear. Generally used for scrolling purposes, the left arrow appears in desktop directory windows.

**G_RArrow** ($400) Causes the right pointing arrow on the right side of the window to appear. Generally used for scrolling purposes, the right arrow appears in desktop directory windows.

*NEW_WINDOW PARAMETERS ( Continued )*

**THE FLAG PARAMETER:**

**G_HSlide** ($800) Causes the horizontal slider on the bottom of the window to appear. Generally used for scrolling purposes, the slider appears in desktop directory windows.

**G_All** ($FEF) Use this name to create a window with all standard characteristics except the information line. This is the most common type of window.

**THE WINDOW NAME PARAMETER:**

The *window name* variable is the title centered in the bar at the top of a window. It's a string, but it is somewhat special:

1. It has a maximum size, so we've given it its own type name, Window_Title.

2. It's passed as a VAR parameter. The GEM/Pascal library will modify it by converting it to the 'C' string format used by GEM.
**SEE:** *Set_WName*, page 7-78.

3. Its address is passed to GEM, so it must be a global variable to ensure that it doesn't disappear when GEM starts using it.

**NOTE:** Window names look better if they have a space or two on each end.

The *x,y,w,h* parameters specify the window's size and position limits. The *x,y* pair limits the window's top left corner position, and the *w,h* pair limits its width and height.

**ALSO:** It's easy to specify illegal values for these numbers, particularly for programs which will run in various screen resolutions, so the GEM/Pascal library gives you an easy way to ensure that the maximum window will fit in the space available: specify a width or height of *0*, and the window will automatically be sized to fit the available workspace in the current resolution. The *x,y* values are then ignored, so they also can be given as *0,0.*

## OPEN_WINDOW ()

```
PROCEDURE Open_Window (handle : Short_Integer ;
                       x,y,
                       w,h : Short_Integer ) ;
```

**EXAMPLES:**

```
Open_Window ( big_window, 0,0,0,0 ) ;

Open_Window ( small_window, 300,100,100,50 ) ;
```

Open_Window draws a window on the screen, and sets it up to receive text and graphics data.

Open_Window requires a valid *window pointer*, previously obtained by *New_Window*, as its first parameter. In GEM parlance, this pointer is often called a *handle*. GEM requires the handle to figure out which window you are using.

The *x,y,w,h* parameters set the initial values for the *x* and *y* coordinates of the top left corner of the window and its width and height. You should not exceed the values you used with *New_Window*. As with *New_Window*, you can let the GEM/Pascal library give you a maximum size window by specifying either a width or height of zero.

**NOTE:** When you open a window with this call, it is always placed in front of all other windows in the display.

**CAUTION:** If the window with the given handle is already open, Open_Window will have unspecified, but unpleasant, results.

## SET_WNAME ()

```
PROCEDURE Set_WName ( handle : Short_Integer ;
                      VAR title : Window_Title ) ;
```

**EXAMPLE:**

```
Set_WName ( big_window, bigger_name ) ;
```

There are occasions when a program might need to change a window's title. This is done easily by passing Set_WName the window's handle along with a new name. The window name is a string. Since it has a maximum size, we've given it its own type name, *Window_Title.*

**CAUTION:** The title is passed as a VAR parameter. The GEM/Pascal library will modify its contents. Since its address is passed to GEM, it should be a global variable so it doesn't disappear when GEM starts using it. *Set_WName* converts the title variable to the *C language* string format required by GEM. As a result, the application program can't simply modify it as originally passed to *New_Window*. To replace the contents of the name variable by assignment, you must either follow the assignment by *Set_WName*, or use *P_To_C_String* to modify the title to suit GEM.
**SEE:** *P_To_C_Str, C_To_P_Str*, page 7-79.

**ALSO:** The *W_Name* bit in the window flags must be set to 1 or *Set_WName* will fail.

**FURTHER:** Window names look better if they have a space character or two on each end.

## SET_WINFO ()

```
PROCEDURE Set_WInfo ( handle : Short_Integer ;
                      VAR info : Window_Title ) ;
```

**EXAMPLE:**

```
Set_WInfo ( big_window, big_info_line ) ;
```

f the W_Info bit in the *New_Window* flags word was set to 1, Set_WInfo allows the application program to write a line in the window's information box.

The string containing window information has a maximum size, so we use the already established type name, *Window_Title.*

**NOTE:** The window information is passed as a VAR parameter. The GEM/Pascal library will modify its contents. Since its address is passed to GEM, it should be a global variable so it doesn't disappear when GEM starts using it. *Set_WInfo* converts the title variable to the *C language* string format required by GEM. As a result, the application program can't simply modify it as originally passed to *New_Window*. To replace the contents of the name variable by assignment, you must either follow the assignment by *Set_WInfo*, or use *P_To_C_Str* to modify the title so as to keep GEM happy.

**SEE:** *P_To_C_Str, C_To_P_Str,* page 6-169.

## SET_WINDOW ()
## GET_WINDOW

Set_Window ()
i := Get_Window

PROCEDURE Set_Window ( handle : Short_Integer )
FUNCTION Get_Window : Short_Integer ;

**EXAMPLES:**

Set_Window ( big_window ) ;

current_window := Get_Window ;
IF Get_Window = small_window THEN ...

These routines are a logically complementary pair. Set_Window makes the specified window the current one for all graphics and text output.

**NOTE:** The current window is not necessarily the front one.

If an application program needs to know which window is currently receiving output, it can find out by calling Get_Window, which returns the handle of the current output window.

The use of *Set_Window* has one other important consequence. If called with a window handle other than zero, then the upper left corner of that window becomes the logical origin for all graphics positions, as described in the section on *Window Text and Graphics*. If the window handle is zero, then the logical origin is the physical screen origin, and the *0,0* point is the upper left corner of the screen.

## WORK_RECT ()

```
PROCEDURE Work_Rect ( handle : Short_Integer ;
                      VAR x,y,
                      w,h : Short_Integer ) ;
```

**EXAMPLES:**

```
Work_Rect ( small_window, xs,ys,ws,hs ) ;

Work_Rect ( 0, xmax,max,wmax,hmax ) ;
```

Work_Rect finds the size of a window's working area rectangle, where graphics and text output take place. The first parameter is a *window handle* previously obtained by *New_Window*. Work_Rect returns the *absolute x,y* coordinates of the upper left corner and the dimensions of the rectangle, *w,h*, relative to the upper left corner of the screen. There are two major occasions to use this call:

1. When drawing within a window, an application program must tell GEM the limits of the part of the screen that it wants to use. It does this by calling Set_Clip and passing it the x,y,w,h values for the current window. Although GEM messages may pass these values to your program, you may also need to use Work_Rect to get them.

2. If the window handle is zero, the returned values represent the entire desktop, less the menu bar and any enclosing box. This is an easy way for an application program to determine how many pixels it has to work with, both horizontally and vertically.

To illustrate, consider the following routine to set the clipping area to the size of the work area of the current window:

```
PROCEDURE clip_cur_window ;

VAR

  x,y,w,h : Short_Integer ;

BEGIN

  Work_Rect ( Get_Window, x,y,w,h ) ;

  Set_Clip ( x,y,w,h ) ;

END
```

**CAUTION:** You should only use this call as-is if the current window is the front one.

**NOTE:** New_Window and Open_Window use Work_Rect internally when you request a maximum size window by passing a width or height of zero.

## BORDER_RECT ()

```
PROCEDURE Border_Rect ( handle : Short_Integer ;
                        VAR x,y,
                        w,h : Short_Integer ) ;
```

**EXAMPLES:**

```
Border_Rect ( 0, xscr,yscr,wscr,hscr ) ;

Border_Rect ( small_window, xs,ys,ws,hs ) ;
```

Border_Rect finds the size of a window's border area rectangle. The first parameter is a window handle previously obtained by *New_Window*. Border_Rect returns the absolute *x,y* coordinates of the top, left corner, and the dimensions, *w,h*, of the entire window, not just the work area. These are relative to the top left corner of the screen.

Programs seldom need to know about the full size of a window. However, when an application program *moves* a window, it does so with *Set_WSize*, which *does* need the size of the full window. Although GEM messages usually pass these values to your program, you can also use *Border_Rect* to get them. For example, this routine is an easy way to move a window to a new *x,y* location without disturbing its size:

```
PROCEDURE move_window ( newx, newy
                  : Short_Integer ) ;

VAR
  cur_window : Short_Integer ;
  x,y,w,h : Short_Integer ;

BEGIN

  cur_window = Get_Window ;
  Border_Rect ( cur_window, x,y,w,h ) ;
  Set_WSize ( cur_window, newx,newy,w,h ) ;

END ;
```

**SEE:** *Get_Window*, p. 7-82 *Set_WSize*, p. 7-89.

BLANK
PAGE

## SET_WSIZE ()

```
PROCEDURE Set_WSize ( handle : Short_Integer ;
                      x,y,
                      w,h : Short_Integer ) ;
```

**EXAMPLE:**

```
Set_WSize ( small_window, 200,100,100,50 ) ;
```

After a window has been created and opened, you might want to change its size, move it to a new position on the screen, or both. These actions are usually taken in response to a mouse event, but a program can move or resize a window any time it is necessary.

Set_WSize requires a valid window handle obtained by New_Window, the *x,y* coordinates of the top left corner, and the dimensions, *w,h,* of a window. GEM will then move or resize the window as requested.

**NOTE:** These actions may, in turn, cause GEM to ask your program for a full or partial redraw of the window.
**SEE:** *Event Management*, page 7-133.

The example program in the discussion of *Border_Rect* shows how easily this very powerful routine is used.

## SET_CLIP ()

```
PROCEDURE Set_Clip ( x,y,
                     w,h : Short_Integer ) ;
```

**EXAMPLE:**

```
Set_Clip ( 302,110,90,40 ) ;
```

A full understanding of the uses of Set_Clip requires an understanding of *Event Management.* For now, it is enough to note that GEM doesn't automatically restrict text and graphic output to the working area of the current window. It's your program's responsibility to tell GEM what boundaries it wants by passing Set_Clip the coordinates of the upper left corner, and the dimensions of a rectangle which defines the limits of the currently available output area.

Once *Set_Clip* has been called, GEM acts fairly intelligently. You can draw lines and shapes which extend off the edge of the clipped rectangle and GEM will take care to ensure that only the appropriate portions are displayed. For text which is placed outside the bounds, GEM is also fairly fast, since drawing text is time consuming while ignoring it is much easier. The significance of this will becomes obvious in the *Event Management* section, where *redraws* are covered.

For an example of a typical use of *Set_Clip*, see the sample program in the previous discussion of Work_Rect.

**NOTE:** The pixel coordinates used with this procedure are always absolute screen coordinates.

**BRING_TO_FRONT ()**
**FRONT_WINDOW**

i := Front_Window

PROCEDURE Bring_To_Front ( handle
:Short_Integer )

FUNCTION Front_Window : Short_Integer ;

**EXAMPLES:**

```
Bring_To_Front ( big_window ) ;
current_window := Front_Window ;
IF Front_Window = small_window THEN ...
```

These routines are a logically complementary pair. Bring_To_Front will make the specified window the front one on the screen. This doesn't say anything about which is the current graphics output window. It does impact the display of a title bar in the specified window.

On the other hand, if an application program needs to know which window is currently the front one, it can find out by calling Front_Window, which returns the *handle* of the window currently in front.

**NOTE:** When a window is opened via *Open_Window,* it is always placed in the front.

**ALSO:** Moving a window to the front may require that parts of other windows be redrawn if they were hidden by the window in its prior position The subject of redrawing windows is discussed in the Event Management section.

If your window is not the front window, you can call *Bring_To_Front* to redraw it, but this is not the usual way to perform a redraw.

## CLOSE_WINDOW ()

PROCEDURE Close_Window (handle : Short_Integer)

**EXAMPLE:**

Close_Window ( big_window ) ;

When a program no longer needs to display a given window, it can erase it from the screen by passing its handle to Close_Window. The window can be displayed again using *Open_Window.*

**NOTE:** Do not close a window which is already closed.

**ALSO:** Closing a window may require that parts of other windows be redrawn if they were hidden by the window. The subject of redrawing windows is discussed in the *Event Management* section.

**CAUTION:** Closing the top (front) window does not cause GEM to send a *Topped* message! If you close the top window, you must perform a *Front_Window* to find the handle of the new front window. Worse: thanks to a bug in Atari GEM, you must then perform a *Bring_To_Front* using that same window handle to *really* make it the top window.

## DELETE_WINDOW ()

PROCEDURE Delete_Window (handle: Short_Integer)

**EXAMPLE:**

```
Delete_Window ( big_window ) ;
```

When a program no longer needs a given window, it can remove it from GEM's cognizance by passing its handle to Delete_Window. The window can't be displayed again by calling to *Open_Window.* It must be recreated by *New_Window* first. Then it is a completely new window as far as GEM is concerned.

**NOTE:** Do not delete a window which is already deleted or which was never created.

**ALSO:** Do not delete a window which is not closed.

BLANK
PAGE

# WINDOW TEXT AND GRAPHICS

Once a program has put a window on the screen, it can print text or draw graphics within it. Keep in mind that GEM doesn't usually remember what a particular window contains. Many actions in GEM can cause the contents of a window to be invalid:

A window is covered by a dialog box, a desktop accessory, or another window, and then uncovered; a window is moved by the user off the screen and then back on; a window is expanded beyond the bounds of what has been drawn. In these situations, GEM knows that the contents of the window are no longer correct and will ask the application program to redraw all or part of the windows.

The concepts of messages, events, and redraws will be covered in the section on Event Management, but as you read this section remember that anything a program draws may need to be redrawn; it is not enough to draw something on the screen and expect it to remain untouched. Your program must remember enough information to recreate the screen. Keep this in mind as you learn about the available window output routines.
**SEE:** *Save_Scrn* and *Restr_Scrn*, page 7-194.

Finally, recall that any and all text and graphic output is limited to the area of the screen within the rectangle defined via Set_Clip. It is not an error to attempt to place objects outside this rectangle; they just won't appear on the screen.

## SET_COLOR ()

```
PROCEDURE Set_Color ( register : Color_Reg ;
                      red,
                      green,
                      blue : Short_Integer )
```

**EXAMPLE:**

```
puce := 7 ;

Set_Color ( puce, 800,100,600 ) ;
```

The Atari ST computer has 16 color registers. Before putting text or graphics onto the screen, you can specify a color register number, 0 through 15, and your output will be displayed in the register's color.

Any color register number can be used to represent any of 512 different colors. As a result, names given to color register numbers mean little, so we refer to them by color register number. However, until you use Set_Color, the default color names are fairly accurate, so we've put them in square brackets alongside the corresponding color register number. The number of color registers available depends on the current screen resolution.

Monochrome monitors have only two color registers available and their names and colors are:

| | |
|---|---|
| **0 [White]** | **1 [Black]** |

High resolution color displays allow four different colors. Their default names are:

| | |
|---|---|
| **0 [White]** | **2 [Red]** |
| **1 [Black]** | **3 [Green]** |

Low resolution color allows 16 color registers:

| | |
|---|---|
| **0 [White]** | **8 [L_White]** |
| **1 [Black]** | **9 [L_Black]** |
| **2 [Red]** | **10 [L_Red]** |
| **3 [Green]** | **11 [L_Green]** |
| **4 [Blue]** | **12 [L_Blue]** |
| **5 [Cyan]** | **13 [L_Cyan]** |
| **6 [Yellow]** | **14 [L_Yellow]** |
| **7 [Magenta]** | **15 [L_Magenta]** |

To take advantage of the ST's color palette, call *Set_Color*, specifying a color register number as its first parameter.

The other three parameters specify an intensity of 0 to 7 for each of the three color guns in the color monitor; red, green, and blue.

GEM is designed to work on future computers as well as the present ST machines, so *Set_Color* asks for an intensity of 0 through 1000. On the ST, GEM converts this 0 to 1000 range into the hardware's 0 to 7 range. Each intensity change of 125 GEM units represents a change of 1 unit to the hardware. This means, for example, that GEM intensity 307 is indistinguishable from intensity 311. Finding the correct GEM intensity level for a particular screen color may require some experimentation.

**TEXT_COLOR ()**
**LINE_COLOR ()**
**PAINT_COLOR ()**

PROCEDURE Text_Color ( color : Color_Reg )

PROCEDURE Line_Color ( color : Color_Reg )

PROCEDURE Paint_Color ( color : Color_Reg )

**EXAMPLES:**

```
Text_Color ( Red ) ;
Line_Color ( Black ) ;
Paint_Color ( Icolor | 8 ) ;
```

These procedures select the colors for text, lines and painted areas. The color numbers passed to these procedures are actually color register numbers.
**SEE:** *Set_Color*, page 7-98.

Line_Color selects the color register for all line drawing calls, including the *Frame_* routines

## TEXT_STYLE ()

PROCEDURE Text_Style ( style : Short_Integer )

**EXAMPLE:**

```
Text_Style ( Slanted | Thickened ) ;
```

Text_Style selects the style for text displayed by subsequent *Draw_String* procedures. The style parameter is an integer number whose bits each control one quality of the text to be displayed. The GEM/Pascal library names and hexadecimal values for these bits are given below. You can create your own names for styles which use a combination of these bits.

**Normal** ($00) Use the normal, upright font. This is the default type.

**Thickened** ($01) The font is made bold-faced. Since this is done by simply increasing the number of pixels displayed, some characters don't look their best when thickened. You might have to experiment to get good-looking results.

**Lightened** ($02) The characters are displayed as ghost characters. This is done by removing pixels from the standard font, so some characters may not look right in this style. Note that this is the style used in the pull-down menus for de-selected menu items.

**Slanted** ($04) The normal font is slanted somewhat, as if the text were italicized.

**CAUTION:** Leave at least one space between slanted text and normal text or weird-looking characters may occur.

**Underlined** ($08) All subsequent text is underlined until this bit is reset.

**Outlined** ($10) Characters are given an outlined appearance. Most characters in the standard font look ok in this style, but once again experimentation may be advisable.

**Shadowed** ($20) Similar to bold (Thickened), but characters are also extended lower. Few characters look good in this style, so it may be of little use to you.

**NOTE:** Some of the above styles are obviously incompatible. The action of GEM when two or more incompatible styles are selected at the same time is not specified.

## TEXT_HEIGHT ()

PROCEDURE Text_Height ( height : Short_Integer );

Besides the character styles described in the previous pages, GEM allows you to select a few other text characteristics. Perhaps the most useful of these is a simple method of selecting the height of your text.

Text_Height takes a single parameter, the desired size of the text. In theory, valid heights range from 1 to 99 or more. In practice, there are only two ranges that produce usable results:

6 through 12 -- Gem uses its "small" font to draw successively larger characters.

13 through 26 -- Gem uses its "large" font to draw successively larger characters.

Text_Height(13) actually draws considerably smaller characters than does Text_Height(12), but with slight differences in appearance.

NOTE: Heights from 1 to 5 produce unreadable small characters (though 5 might be usable if all capital letters, with no special text style, are used). Heights above 26 produce results identical with those of Text_Height(26).

## DRAW_MODE ()

PROCEDURE Draw_Mode ( mode : Short_Integer )

### EXAMPLE:

```
Draw_Mode (4) ;
```

Draw_Mode selects the manner in which text and graphics are drawn. there are four modes, 1 through 4, and their meanings are:

| Mode | Meaning |
|---|---|
| 1 | **Replace** Completely replace any existing background. This is almost always the mode used for non-text drawing. |
| 2 | **Transparent** The *on* bits in graphics objects are OR-ed with the background. For example, allows diacritical marks to be drawn over characters. |
| 3 | **XOR** The *on* bits in the object are *XOR*-ed with the background. Normally only used with a solid background. |
| 4 | **Reverse transparent** The *off*, or *0*, bits in drawn objects are *OR*-ed with the background. Again normally used only with a solid background to produce reverse video images. |

## LINE_STYLE ()

PROCEDURE Line_Style ( lstyle : Short_Integer )

EXAMPLE:

Line_Style ( 5 ) ;

This procedure determines the line style used by subsequent line-oriented procedures. Valid style numbers are 1 through 6.

Style one requests the default solid line, while increasing numbers ask for dashed and dotted lines of various patterns. The following chart illustrates the pattern density of each style. The *x* characters in the patterns each represent a pixel. The patterns repeat every 16 pixels.

| Style | Pattern | Description |
|---|---|---|
| 1 | xxxxxxxxxxxxxxxx | solid line |
| 2 | xxxxxxxxxxxx | long dash |
| 3 | xxx xxx | dots |
| 4 | xxxxxxx xxx | dash & dot |
| 5 | xxxxxxxx | dash |
| 6 | xxxx xx xx | dash, dot, dot |

**NOTE:** Setting a line width greater than 1 causes Line_Style to be ignored

## PAINT_STYLE ()

PROCEDURE Paint_Style ( pstyle : Short_Integer )

**EXAMPLE:**

```
Paint_Style ( 27 ) ;
```

Paint_Style selects the pattern used by subsequent calls to object-oriented paint routines. The numbers 0 through 35 are valid style numbers. The style numbers have the meanings shown on the next page:

| Style | Description |
|---|---|
| 0 | **No paint.** The area is not painted at all. For routines of the form *Paint_*, this style produces the same result as if the corresponding *Frame_* routine had been called. |
| 1 | **Solid.** The entire area will be completely filled with the *Paint_Color* specified. |
| 2 - 25 | Various "dither" patterns. If painted on a solid background of another color, these patterns can produce the appearance of more colors than are otherwise available. Possibly very useful when used in high resolution color. In monochrome, these patterns offer a means of displaying objects on a gray scale of varying intensity. In monochrome mode, the GEM desktop is filled with pattern 5. |
| **NOTE:** | Pattern 9 has the same effect as style 1; solid paint. |
| 26 to 37 | Various cross-hatch patterns. These patterns don't produce the subtle effects that some of the 2 through 25 patterns do, but they do make striking and distinctive backgrounds. |

## PAINT_OUTLINE ()

PROCEDURE Paint_Outline ( flag : Boolean )

EXAMPLE:

```
Paint_Outline ( True ) ;
```

Passing Paint_Outline a True value causes areas painted by subsequent *Paint_* calls to have solid outlined borders. A patterned area will thus have a solid boundary, rather than a ragged outline. False produces no outline.

**NOTE:** The *True* state is only meaningful if the *Paint_* Pattern is something other than 1; *solid.*

**COMMENT:** If you want to have a painted shape of one color outlined with a line of another color, use a *Paint_* call for the interior color followed by *Line_Color* and a *Frame_* call to produce the complementary outline.

## DRAW_STRING ()

PROCEDURE Draw_String ( x,y : Short_Integer ;

text : Str255 )

**EXAMPLE:**

```
Draw_String (10,10,'Start near the upper left corner') ;
```

Draw_String displays the specified text string in a horizontal line starting at the *x,y* pixel coordinates, relative to the upper left corner of the current output widow. The text color and style previously selected will be used to draw the characters.

**NOTE:** GEM actually draws each character of the string, placing individual bits in the appropriate locations in screen memory. In the case of color text, this can involve setting bits in several of the color planes, but the entire process is transparent when you use Draw_String.

**MOVE_TO ()**
**PLOT ()**

PROCEDURE Move_To ( x,y : Short_Integer )

PROCEDURE Plot ( x,y : Short_Integer )

**EXAMPLE:**

```
Move_To ( 0,0 ) ;
Plot ( curx+1,cury-1 ) ;
```

These two procedures accomplish nearly the same thing: they move an invisible graphics cursor to a location within the current graphics window. The coordinates of the new cursor location are specified by two parameters, the horizontal and vertical positions, *x* and *y*. The numbers given are pixel coordinates, relative to the upper left corner of the working area of the current window as previously defined by *Set_Window.*

The difference between these routines is simple: Move_To moves the cursor to the specified location and does *nothing else.* Plot moves the cursor and then draws a single pixel, using the color register last specified by *Line_Color.*

**NOTE:** If the pixel specified is outside the current clipping rectangle it is not painted, but the cursor is moved.
**SEE:** *Set_Clip*, page7-90.

**ALSO:** The "invisible cursor" only refers to points used in *Move_To, Plot, Line, and Line_To.*

**COMMENT:** It isn't illegal to specify a pixel location outside the bounds of either the clipping rectangle, the window, or even the entire screen. This comment applies to all the routines in this section. In fact, with some routines, such as *Paint_Arc*, you can often get meaningful results only by giving points far outside the displayable area.

**LINE ()**
**LINE_TO ()**

```
PROCEDURE Line ( from_x,from_y,
                 to_x, to_y : Short_Integer )

PROCEDURE Line_To ( from_x,from_y,
                    to_x, to_y : Short_Integer )
```

**EXAMPLES:**

```
Line ( 10,10, 40,10 ) ;
Line_To ( 40,40 ) ;
Line_To ( 10,40 ) ;
Line_To ( 10,10 ) ;
```

These procedures are paired for good reason. Each draws a straight line between a pair of specified pixel endpoints. The pixel coordinates given are usually relative to the upper left corner of the current window.
**SEE:** *Set_Window*, page 7-82.

When each is finished, the invisible graphics cursor is positioned at the last pixel drawn. The color register specified in the last call to *Line_Color* is used for all pixels in a line.

The difference between these routines is that *Line* specifies the coordinates of both ends of the line. *Line_To* assumes that the line will begin at the current graphics cursor location and end at the specified x,y pair. The four lines of the example illustrate these ideas: combined, they draw a square 30 pixels on a side.

If any portion of a line to be drawn is outside the current clipping rectangle, that portion is not actually drawn, but the location of the invisible graphics cursor is still updated as described.
**SEE:** *Set_Clip*, page 7-90.

**NOTE**: The "invisible cursor" only refers to points used in *Move_To, Plot, Line, and Line_To.*

**PAINT_RECT ()**
**FRAME_RECT ()**
**PAINT_ROUND_RECT ()**
**FRAME_ROUND_RECT ()**

```
PROCEDURE Paint_Rect ( x,y,
                       w,h : Short_Integer )

PROCEDURE Frame_Rect ( x,y,
                       w,h : Short_Integer )

PROCEDURE Paint_Round_Rect ( x,y,
                       w,h : Short_Integer )

PROCEDURE Frame_Round_Rect ( x,y,
                       w,h : Short_Integer )
```

**EXAMPLE:**

```
Paint_Rect ( 41,10, 100,30 ) ;

Frame_Rect ( 10,10, 30,30 ) ;

Paint_Round_Rect ( 41,10, 100,30 ) ;

Frame_Round_Rect ( 10,10, 30,30 ) ;
```

Paint_Rect paints a rectangle with its top left corner at the pixel coordinates specified by the *x,y* pair.
**SEE:** *Set_Window.*

The dimensions of the rectangle are given by the *w,h* pair, in pixels. The interior of the specified rectangle will be filled with the color from the color register number last specified by *Paint_Color.* The last call to *Paint_Style* determines the manner in which the color is painted, and the setting of the *Paint_Outline* flag determines whether or not a solid border is drawn. Pixels outside the bounds of the current clipping rectangle will not be painted or drawn.
**SEE:** *Set_Clip*, page 7-90.

*Frame_Rect* has the same effect as calling *Paint_Rect* after setting *Paint_Style (0)* and *Paint_Outline (True)*, except that *Frame_Rect* uses the current *Line_Style* and *Line_Color.*

*Paint_Round_Rect* and *Frame_Round_Rect* are used in the same way as *Paint_Rect* and *Frame_Rect*, and with the same parameters. The difference is that the resulting rectangles have rounded corners. GEM chooses the radii of the corners, so for really exacting drawings we recommend combining ordinary rectangles and arcs.

**PAINT_OVAL ()**
**FRAME_OVAL ()**

```
PROCEDURE Paint_Oval (x_center,y_center,
                      x_radius,y_radius : Short_Integer )

PROCEDURE Frame_Oval (x_center,y_center,
                      x_radius,y_radius : Short_Integer )
```

**EXAMPLES:**

```
Paint_Oval ( 160,150, 100,30 ) ;
Frame_Oval ( 100,100, 60,60 ) ;
```

*Paint_Oval* paints an oval with its center at the pixel coordinates specified by its first two parameters, usually relative to the current window.
**SEE:** *Set_Window*, page 7-82.

The size of the oval is determined by the second pair of parameters, again in pixel units. The eccentricity of the oval is determined by the relative difference of the two radii. In high resolution and low resolution, specifying identical radii causes *Paint_Oval* to draw a circle. This is not the case for medium resolution, however.

The interior of the specified oval will be filled with the color of the last color register number specified by *Paint_Color*. The last call to *Paint_Style* determines the manner in which the paint color is applied, and the setting of the *Paint_Outline* flag determines whether or not the oval has a solid border. Pixels outside the bounds of the current clipping rectangle will not be painted or drawn.
**SEE:** *Set_Clip*, page 7-90.

*Frame_Oval* produces the outline of the specified ellipse, using the last specified *Line_Style* and *Line_Color*.

**FRAME_ARC ()**
**PAINT_ARC ()**

```
PROCEDURE Frame_Arc (x_center,y_center,
                     x_radius,y_radius,
                     start_angle,end_angle : Short_Integer )

PROCEDURE Paint_Arc (x_center,y_center,
                     x_radius,y_radius,
                     start_angle,end_angle : Short_Integer )
```

**EXAMPLE:**

```
Frame_Arc ( 100,100, 60,60, 0,900 ) ;
Paint_Arc ( 160,150, 100,30, 2700,3600 ) ;
```

Paint_Arc and Frame_Arc are similar to *Paint_Oval* and *Frame_Oval.* The specifications for the center point coordinates and radii are identical, as are the effects of specifying *Paint_Color, Paint_Style, Paint_Outline, Line_Style,* and *Line_Color.*

The *_Arc* routines have two additional parameters: *start_angle* and *end_angle*. These numbers determine where the arc will begin and end. The angles are given in units of *tenths of a degree*. There are *3600 angle-units* in an arc which makes a full ellipse. The origin of angles, *0 degrees* is the normal mathematic origin, *due east*, and angles are measured *counter-clockwise* from that position. The result of using values in excess of 3600 units is undefined.

**NOTE:** When you use *Paint_Arc, Paint_Outline* affects both the arc of the specified oval and the radial lines. *Frame_Arc* draws only the arc. If you need an outlined, filled arc without radial lines, first use *Paint_Arc* with the outline turned *off*, followed by *Frame_Arc*. Set *Line_Color* and *Line_Style* to make the arc outline match the interior.

BLANK
PAGE

# MOUSE CONTROL

Although your program has no control over the position of the mouse pointer, most other aspects of the mouse are easily controlled. For example, you can hide the mouse, bring it back, change its appearance, and modify GEM's response to it. GEM's use of the mouse is complex and, because the mouse interacts with so much of GEM, the description of mouse events is delayed until the *Event Management* section, beginning on page 7-133.

**HIDE_MOUSE**
**SHOW_MOUSE**

```
PROCEDURE Hide_Mouse ;
PROCEDURE Show_Mouse ;
```

**EXAMPLE:**

```
Hide_Mouse;
redraw_window( cur_window ) ;

    { redraw_window is for example only }

Show_Mouse;
```

Sometimes it's useful to turn off the mouse pointer. For example, it is a good idea to turn off the mouse before updating a window. It's usually best to hide the mouse when you move a window on the screen, too. That's because the mouse routines in GEM aren't very smart. When you move the mouse, they sometimes restore the screen the way it was before the mouse was moved... even if you've already changed the underlying screen!

One aspect of the *Show_Mouse* and *Hide_Mouse* calls is that they nest to quite a depth. If you call *Show_Mouse* three times, you must call *Hide_Mouse* three times before the mouse will disappear. Likewise, if you call *Hide_Mouse* twice, it takes a pair of *Show_Mouse* calls to make the mouse visible again. The effect is that of a counter which can take on either positive or negative values: as long as the counter is zero or positive, the mouse is visible. The advantage of this feature is that a routine can hide and then restore the mouse without having to worry about whether its caller actually wanted the mouse restored or not.

Certain system actions take precedence over the mouse's visibility. For example, any time an alert box is displayed, the mouse pointer is made visible and usable for the duration of the box's activity. The same is true of dialog boxes. When these boxes are terminated, the state of the mouse prior to the box call is restored.

**INIT_MOUSE**
**SET_MOUSE ()**

```
PROCEDURE Init_Mouse ;

PROCEDURE Set_Mouse ( shape : Mouse_Type ) ;
```

**EXAMPLE:**

```
IF user_error <> 0 THEN Init_Mouse ...
Set_Mouse( M_Bee ) ; { tell user we're busy }
```

When your program needs to ensure that the mouse is visible and active, the best choice is usually a call to Init_Mouse. This routine performs two important actions:

**1.** It resets the show/hide mouse counter to zero, causing the mouse to be displayed and guaranteeing that a single *Hide_Mouse* call will turn it off.

**2.** It sets the mouse shape to that of the arrow, *M_Arrow*, the most usual form of mouse on ST computers.

If your program needs another shape for the mouse, GEM provides a total of 8 pre-defined forms. To choose one, simply pass the appropriate shape name to *Set_Mouse.*

The standard mouse shapes are:

**M_Arrow** The default form. An arrow pointing toward the upper left. Outlined for better visibility.

**M_Text_Curs** A vertical bar with flared ends. Often used as a cursor for text processing.

**M_Bee** The *busy bee*, usually used to indicate that the system will not react to mouse clicks.

**M_Point_Hand** A hand with the index finger outstretched, pointing toward the upper left.

**M_Flat_Hand** A hand with all fingers outstretched, often used to "lift" a shape and carry it to another part of the screen.

**M_Thin_Cross** Very thin cross-hairs. Useful for pointing to exact pixels in graphics programs.

**M_Thick_Cross** Thicker cross hairs. Probably more suited to text pointers.

**M_OutIn_Cross** An outline cross hair. Much like taking a very thick cross-hair and removing the center hairs from it.

**NOTE:** Changing the mouse pointer's shape does not change its visibility. Only *Init_Mouse*, *Hide_Mouse*, and *Show_Mouse* will affect the visibility of the mouse.

## SET_MFORM ()

PROCEDURE Set_MForm (VAR form: Mouse_Form) ;

**EXAMPLE:**

Set_MForm ( angle_form ) ;

When none of the predefined mouse forms is exactly what you want, define your own shape by calling Set_MForm with a filled-in *Mouse_Form* record.

A mouse form is always 16 pixels wide by 16 pixels high. GEM represents it as 16 integers. In addition to a shape, a mouse can have a mask, usually an outline, so the shape will stand out on any color background. You can assign color registers to both the mask and the shape.

There are other essential data, so the *Mouse_Form* actually consists of 37 integers. For clarity and consistency, it has a record structure. You can inspect it in the file *GEMTYPE.PAS* on your Personal Pascal diskette. The various fields and their contents are named and described on the following page:

## MOUSE_FORM FIELDS:

**hot_x** Gives the horizontal pixel coordinate of the "hot spot" in the mouse form. The screen location of this hot spot is what GEM passes to your program when a mouse position event occurs. The coordinate is relative to the left side of the mouse shape, so it must be in the range 0 through 15.

**hot_y** The vertical pixel coordinate of the "hot spot." It, too, must be in range of 0 through 15.

**res** Resolution of the mouse. Under GEM on an ST computer, this value must be 1.

**mask_color** Color register (0 to 15) for the outline mask. Should not be the same as *data_color.* Be careful about assigning color register numbers if your program is to run on both color and monochrome systems.

**data_color** Color register for the shape. Each bit of the shape will be displayed using this color register value. Read the remarks in *mask_color.*

**mask** A 16-element integer array, in which each bit set to one corresponds to a pixel to be displayed in the color specified by mask_color.

**data** A 16-element integer array, in which each bit set to one corresponds to a pixel which will be given data_color.

This program fragment is only an example; it will not compile as written:

```
m : Mouse_Form ; { a VAR declaration }

word : integer ;

...    { code in program or subroutine: }

  m.hot_x := 2 ;  m.hot_y := 3  ; { hot spot }
  m.res := 1 ; { must be this value! }
  m.mask_color := White ;
  m.data_color := Black ; { usual }

  FOR word := 0 TO 15 DO

  BEGIN      { clear out both arrays }
    m.Mask[word] := 0 ;  m.Data[word] = 0 ;
  END

  m.data[0]:=$70E7;  m.data[1]:=$8908;
  m.data[2]:=$8908;  m.data[3]:=$89EF;
  m.data[4]:=$8821;  m.data[5]:=$8821;
  m.data[6]=$71CE;

Set_MForm( m ) ;
```

## BEGIN_MOUSE
## END_MOUSE

PROCEDURE Begin_Mouse ;

PROCEDURE End_Mouse ;

After an application program calls Begin_Mouse, and until it calls End_Mouse, the mouse becomes disabled as far as most of GEM is concerned. In particular, clicks in the menu bar and in other windows are ignored. Most regular GEM operations are suspended. However, the program that performed the Begin_Mouse can still get mouse and button events, so it can continue to use the mouse.

Begin_Mouse is a kind of last resort call. A program or subroutine might use it to guarantee that nothing can happen on the screen that it doesn't completely control. For example, a graphics drawing program might want to ensure that no desk accessory would be enabled, since an accessory would destroy the screen. Once again, the concept of mouse management is tightly tied to event management. Be sure to read the section on Event Management.

BLANK
PAGE

# EVENT MANAGEMENT

We've referred to *Event Management* in many of the preceding sections. You will have become a truly GEM-oriented programmer when write your programs with this important concept in mind. First, we'll discuss generalities, then we'll get down to some actual cases.

## TASKS UNDER GEM

GEM is a limited multi-task system. Even while our program is deep in the throes of solving some simultaneous equations or blasting wart-hogs, the user can move the mouse to any portion of the screen, perhaps the menu bar and then to a menu item, or perhaps a desk accessory.

In a more complicated operating environment, these user actions might immediately cause a *task switch* and activate a program or routine such as an accessory program. That program might actually consist of several programs, one for each event which the mouse pointer or button clicks could activate. Each program might have its own ideas about screen priorities and what to do about other programs. Writing even a simple application program in such an environment can be quite a job. Besides learning all the GEM graphic-oriented calls we have discussed so far, you would have to understand topics such as queues, interrupts, scheduling, task activation and suspension, and much more. And, for the average application, it is not clear that all that extra work buys much.

Enter, then, Event Management under GEM. We said that GEM was a limited multi-tasking system. In practice, this means that the only tasks which can proceed on an interrupt-driven basis, while the application program has control, are the keyboard, mouse, disk, and other low-level hardware events. GEM, instead of causing some user task to start when one of these events occurs, simply builds an "event queue." Not until the current application program pauses to wait for an event will any action take place. In other words, GEM will not interrupt a currently running program in order to allow another one to execute.

More specifically: If a desk accessory has been installed, *and* if there is an event in the queue which it must process, *and* if it has been waiting for the event, *and* if the application program also pauses to wait for one of its events, *then* and *only then* does GEM allow the accessory to take control away from the application program. This includes the possibility that the user might click the mouse on a desk accessory item in the menu, in which case the accessory is told it was "opened" so that it knows to place its display on the screen.

Excepting for these possible desk accessory actions, all other events in the queue are messages for the application program. Since the application program does not receive these messages until it asks for them, and since there may be several messages in the queue, it is a good idea to process as much of the queue as possible at all times.

Consider the following scenario: the user has requested a desk accessory. When the application program pauses to wait for an event, the desk accessory takes over. The user interacts with the desk accessory, finally finishing with it. The desk accessory tells the application program to clean up the messy screen and exits. The application program, which perhaps thought it was waiting for a key from the keyboard, suddenly finds that it must redraw a major portion of the screen! And just to make life more fun, while the program is redrawing the screen, along comes that key that it was waiting for. Sounds complicated enough, even if it isn't true multitasking. To make matters even more anxious, some accessories allow other accessories to be invoked while they're still running! Fortunately, the OSS GEM/Pascal library has a solution to all this.

# MULTIPLE EVENTS

Within GEM, and accessible via the GEM/Pascal library, there is a single operating system call which allows application programs, including desk accessories, to request information about all possible events. And if, indeed, both a keyboard event and a redraw message are waiting for the program, it can decide which to process first, though normally it would always process messages first.

And just what kinds of events can we wait for? Thought you'd never ask:

**KEYBOARD EVENTS** A keyboard event can occur each time the user presses a key on the keyboard.

**MOUSE BUTTON EVENTS** Depending upon a program's event request, a button event might consist of pushing the mouse button, releasing it, waiting for both push and release, or even waiting for several such cycles.

**MOUSE POINTER EVENTS** A program can ask GEM to tell it when the mouse pointer enters or leaves either one or both of two rectangles on the screen.

**MESSAGES** We have already discussed some of the reasons for redraw messages. Besides these, a program can wait for messages regarding the slider rectangles, the grow, full, and close boxes, menu items, window movement or expansion, and more.

**TIMER EVENTS** The program can use a timer alone, or in conjunction with other events. For example, by combining a zero-time timer with a keyboard event request, a program can test to see if a key is waiting, without actually halting what it is doing unless the key is hit.

**NOTE:** Timer events are special. They are the only kind of events that can occur even while a desk accessory is active! This feature is useful, as it allows us to get around some of the holes in GEM. More about this later.

On the next page, we've sketched the skeleton of a typical GEM-oriented, event-driven program. Rather than do it in Personal Pascal, we present it in pidgin-Pascal/English to save space. Later, we'll look at parts of an actual Pascal program.

```
LOOP forever while we
    WAIT for any event.
    IF it was a message
        THEN do message processing.
    IF it was a mouse position event
        THEN do position processing.
    IF it was a button click event
        THEN do button processing.
    IF it was a keyboard event
        THEN do keyboard processing.
    IF it was a timer that expired
        AND none of the other events happened
    THEN do timer processing.
ENDLOOP (* and do it forever *)
```

On the face of it, this is an easy program. But note the difference between this and a program which simply waits for a line of input before doing anything. Here, each key in the line is processed separately, and if the user decided to change things and go drag down a menu, we process the request right then and there. The advantage? Suppose that one of the items in the menu bar was *Help.* Imagine how nice this would be: The user is typing in some response, forgets exactly what was needed, moves the mouse cursor to the *Help* item, receives a helping message via an alert box, removes the alert box, and goes on typing the rest of the response! With a properly written GEM program, that's almost a natural.

Most of what we have just discussed is not too difficult to implement: putting up an alert box in response to a menu item request, for example, is downright easy. True, it takes a little work to get the menu up there but, once it is there, the rest is simple. The hardest part of implementing a proper GEM program comes when our application receives a redraw message event.

## WINDOW EVENTS

First, let's examine the situations which will NOT cause GEM to ask us to redraw our window. For starters, alert boxes and drop down-menus, even though they cover part of our screen, never cause redraws. That's because GEM maintains a screen buffer internal to itself which is capable of holding up to one-fourth (25%) of the main screen. The buffer is used only to hold the screen portions obscured by menus and alert boxes. Implications of this include:

**1.** No menu or alert box can exceed the size of that buffer and,

**2.** GEM does not allow both a drop down menu and an alert box on the screen at the same time.

Other user actions causing messages from GEM without involving a redraw might include moving a window, as long as all of it was visible before the move, and resizing a window to make it smaller. In these cases GEM can often modify the screen display without help from the application program.

On the other hand, suppose the user moves a window which had been either partially or fully obscuring another window. Now part of that rear window is visible, but GEM has long since forgotten what was in it. What has to happen? A redraw event in that previously obscured window!

## REDRAW EVENTS

As we learn more about messages, we'll find more circumstances that can force a redraw. For now though, let's take a look at what a redraw involves:

It's not inconceivable that the *rear-most* window on the screen might be obscured by three, four, or more different objects, each in a slightly different position on the screen. The result might be that small pieces of that rear-most window are showing through the others here and there. If it comes time to redraw that rear window, how can it possibly know what portions of itself it needs to redraw?

Partially, the process is automatic, taken care of by GEM. But a large part of the responsibility for this task is given to the application program. On the next page is another pidgin-Pascal/English routine which outlines the complete process of updating a window. This is the routine to implement the "Do message processing" routine from the previous pidgin listing.

```
PROCEDURE Do_Message

  IF message is redraw of window "W"

  THEN BEGIN

    Tell GEM we are beginning the update (redraw).
    Get x,y,w,h of redraw from the message.
    Use First_Rect to find first visible rectangle.

    WHILE some rectangle is visible

    BEGIN
      Find intersection of redraw and visible rect's.
      Redraw window within that intersection (if any).
      Use Next_Rect to find next visible rectangle.
    END

    Tell GEM we finished that update.

  END

  ...    { more... }
```

Aha! There are some new goodies in that pseudo-listing. First, there are two new calls to tell GEM that we are beginning or finishing a screen update (redraw). These are described later, under the names *Begin_Update* and *End_Update.*

*First_Rect* and *Next_Rect* are two more new calls. These will be discussed in detail later, but for now, note that each window has a list of its visible rectangles maintained for it by GEM. These calls enable the Personal Pascal programmer to find those rectangles.

Then there is the line disarmingly titled "find intersection..." Fear not, for the GEM/Pascal library has provided that routine for you also: *Rect_Intersect.* Almost easy so far. The next line is the killer.

"Redraw window within that intersection." After all this discussion about redraw events, we still end up having to make a statement as general as this. Unfortunately, though, this is indeed where we must leave it. Nothing in GEM, beyond the application program, can remember what the program placed in a given window. Is it text? Line drawings? A painted picture? The application program must literally redraw the relevant portion of the screen.

Now if the window contains only text, there is a reasonably workable method of restoring it. Requirements:

**1.** The application program must buffer the text somewhere within itself.

**2.** It must be able to easily re-print the entire window with text from this buffer.

**3.** It must set the clipping rectangle (via *Set_Clip*) to the rectangle which is the intersection of the redraw area and the first or next visible rectangle.

**4.** It must, indeed, redraw the entire window. Thanks to that clipping rectangle and the fact that we repeat this for each visible window, it works. And, because it is text, it is not too slow.

Not too bad. You might wonder about the need to buffer the text at all times. Actually, this is good practice anyway, if you allow resizing of the window. After all, if the user shrinks the window down to one line high and 20 characters long, you would certainly want to be able to fill the window if it is suddenly "grown" to full size. Since, when using *System_Font*, the largest viable text screen is 24 lines by 80 characters, buffering all the text is not too onerous.

But what about graphics? Well, it is not out of the question to reserve a 32K byte chunk of memory which serves as a buffer for the graphics screen. The *Save_Scrn* and *Restr_Scrn* routines make that easy. On the other hand, a better solution might be to find a way to avoid obscuring a graphics window except with the menu or alert boxes.

One way is to avoid putting up a menu. With no menu, there are no desk accessories to be found, as far as the mouse is concerned.

And now, at last, we are ready to actually look at the routines which make up the Event Management support library.

BLANK
PAGE

# EVENT MANAGEMENT ROUTINES

This section presents the routines used for event management. Several of the window management routines are relevant here, as well as several Mouse Control routines. It's helpful to have read the sections pertaining to them before proceeding.

## GET_EVENT ()

```
i := Get_Event ()

FUNCTION Get_Event ( event_mask : Short_Integer ;
                     btn_mask       : Short_Integer ;
                     btn_state      : Short_Integer ;
                     n_clicks       : Short_Integer ;
                     ticks          : Long_Integer ;
                     r1_flag        : Boolean ;
                     r1_x,r1_y,
                     r1_w,r1_h      : Short_Integer ;
                     r2_flag        : Boolean ;
                     r2_x,r2_y,
                     r2_w,r2_h      : Short_Integer ;

                     VAR message  : Message_Buffer ;
                     VAR key      : Short_Integer ;
                     VAR bstate,
                         bcnt     : Short_Integer ;
                     VAR mx, my   : Short_Integer ;
                     VAR kbd_state : Short_Integer )

                     : Short_Integer ;
```

**EXAMPLE:**

```
{ Check to see if a keyboard event is waiting.  If so,
return key code.  If not, return -1.  This routine is
better than using Keypress in GEM programs. }

FUNCTION key_no_wait : Short_Integer ;
  { declarations omitted for brevity }

BEGIN
  event:=Get_Event (E_Keybd | E_Timer,
                    0,0,0, { no button goodies }
                    0,     { timer cnt of zero! }
                    False,0,0,0,0, { no mouse rect's }
                    False,0,0,0,0,
                    msg_area,
                    what_key, { what we really want }
                    dummy,dummy,dummy,
                    dummy,dummy ) ; { unused }

  IF (event & E_Keybd) <> 0 THEN
    key_no_wait := what_key

  ELSE

    key_no_wait := -1 ; { negative value if no key }

END ;
```

# BLANK PAGE

## GET_EVENT PARAMETERS

Just look at all those parameters! The hardest part of using *Get_Event* is figuring out which parameters to use to do what. Yet, as the example illustrates, simple cases can use many dummy parameters. Further, once an application program has coded a call to *Get_Event,* as in the example, that coding can be placed in a subroutine and called by a simple name when needed.

The various parameters and their meanings are described individually on the following pages:

*GET_EVENT PARAMETERS ( Continued )*

**event_mask** is an integer whose bits indicate which kinds of events we are expecting. The bits have the same names and meanings as those returned by the function and are described below.

**btn_mask** is an integer in which only the two least significant bits have meaning. If the least significant bit ($0001) is on, we are waiting for left mouse button events. If the next bit ($0002) is on, we are waiting for the right button. This mask is meaningless if the corresponding bits in the *event_mask* are set to 0.

**CAUTION:** Some versions of Atari GEM have problems recognizing right-button-only events.

**btn_state** is another integer in which only two bits are meaningful. Its bits correspond to those of *btn_mask,* but here a bit which is *on* indicates we're waiting for a *button-down* event. An *off* bit signals a wait for a *button-up* event. These bits are meaningless *if btn_mask* and *event_mask* do not expect a mouse button.

**n_clicks** is an integer which tells GEM how many of the button events specified by *btn_mask* and *btn_state* to wait for.

**ticks** is a long integer which gives the number of milliseconds the caller wants GEM to wait before returning. This number is only meaningful if the *E_Timer* bit in *event_mask* is set. A tick count of *zero* forces an immediate return. As can be seen from the example, this can be useful, since other conditions in the event queue can be returned at the same time.

**r1_flag** is a boolean value which indicates "sense" of a rectangle defined by the following four parameters. If the value is *True*, then the mouse causes an event when it *leaves* the defined rectangle. If *False*, the event happens when the mouse *enters* the rectangle. Only meaningful if the *E_MRect_1* bit in *event_mask* is set.

**r1_x, r1_y, r1_w, r1_h** are standard pixel coordinate words defining a rectangle's upper left corner, *x* and *y*, and its width and height, *w* and *h*. **SEE:** *r1_flag*, above.

**r2_flag** functions the same as *r1_flag*, but allows the program to pass a second rectangle for GEM to monitor. *E_MRect_2* must be set or this rectangle will be ignored.

**r2_x, r2_y, r2_w, r2_h** are standard pixel coordinate words defining a rectangle's upper left corner, *x* and *y*, and its width and height, *w* and *h*.
**SEE:** *r2_flag*, above.

**message** is actually a buffer defined by the type Message_Buffer in the file *GEMTYPE.PAS.* Notice that you must pass a variable, to be modified. The message buffer is described more fully later in this section.

**key** is the code generated when the user presses a key, if keyboard events were enabled by the *E_Keyboard* bit in *event_mask.* This parameter must be passed as a modifiable variable.

**bstate** is another modifiable variable. It returns the state the mouse buttons were in when the event was placed in the queue.

**bcnt** is the count of the number of times the button state has been reached.
**SEE:** *btn_state*, page 7-154.

Unless the timer has expired, this count should be not less than 1 nor more than the count assigned to *n_clicks*, above. Again, a VAR parameter.

**mx** and **my** are integer modifiable variables which receive the current *x* and *y* position of the mouse cursor, in absolute pixel coordinates. The result is meaningful only with *E_Button, E_MRect_1*, or *E_MRect_2* set within the *event_mask* word.

*GET_EVENT PARAMETERS ( Continued )*

**kbd_state** is returned only when a mouse button event is requested. Bits within this word give the state of various keys on the keyboard when the button event is placed on the queue. The following hexadecimal values describe the bits:

$0001 **The right-side shift key.**

$0002 **The left-side shift key.**

$0004 **The Control key.**

$0008 **The Alt key.**

A bit set to 1 implies that the key was **pressed.**

## GET_EVENT RETURN VALUE

The value returned by the *Get_Event* function is another integer whose individual bits means various things. For this function, the meaning of each bit corresponds to the meaning of the same bit in its first parameter, *event_mask*, as described above. If a bit is set to one, GEM is trying to tell our application program that a particular event has occurred. The bits are:

**E_Keyboard** ($01) Keyboard events.

**E_Button** ($02) Mouse button events.

**E_MRect_1** ($04) Mouse into/out of first rectangle.

**E_MRect_2** ($08) Mouse into/out of second rectangle.

**E_Message** ($10) A message in the message buffer.

**E_Timer** ($20) Timer expiration.

If a program is awaiting several types of events, it is usually best to process the message events first. Since the value returned by this function may have any or all of the specified bits set, it is best to assign its value to a variable. You can check for meaningful bits by using the *binary and* operator, *&*.

## COMMENTARY:

When a properly requested keyboard event occurs, the ASCII code for that key is placed in the key variable. The ST can generate key requests which go beyond the ASCII set. The appendix contains a table of all possible encoded key values.

For the mouse button, waiting for a single button-down or button-up event is straightforward. Waiting for multiple clicks may involve adjusting the time count to get meaningful results.

Timer events alone are simple: GEM waits until the specified number of milliseconds have elapsed and then returns. In combination with other event requests, it can make otherwise simple events become important by returning before the user responds, if the timer expires. Recall the earlier note: timer events are the only ones an application program can receive while a desk accessory is active. This allows interesting possibilities, such as accessories that allow other accessories to run....

Finally, the two mouse entry/exit rectangles are fairly self-explanatory. A good use for this event might be knowing when the pointer moves into or out of a window's main work area. If you're displaying text in the window, your program might want to change the mouse into a text cursor when it is moved into the work area.

BLANK
PAGE

## BEGIN_UPDATE
## END_UPDATE

PROCEDURE Begin_Update ;

PROCEDURE End_Update ;

This pair of procedures frames an update sequence. Once you've called Begin_Update, GEM will allow no further change to screen rectangles, such as windows, until the corresponding End_Update is called. This is important!

Assume for a moment that the process of redrawing a particular window is a fairly lengthy process. If GEM didn't freeze the screen during the redraw, the user could request screen changes before the redraw finished. Two processes, perhaps a main program and an accessory, might try to draw the same area of memory at the same time.

## FIRST_RECT ()
## NEXT_RECT ()

```
PROCEDURE First_Rect ( handle : Short_Integer;
                       VAR x,y,
                       w,h : Short_Integer ) ;

PROCEDURE Next_Rect ( handle: Short_Integer;
                      VAR x,y,
                      w,h : Short_Integer ) ;
```

**SEE:** *Rect_Intersect*, page 7-164.

Calls to these routines can be made at any time, but they're most commonly used during a redraw. Within GEM, a window is described partly as a set of non-overlapping rectangles. When the window is in front, the set consists of a single rectangle equivalent to the work area of the window. If the window is completely hidden behind other windows, the set is empty, but for partially obscured windows, the set may contain one or more rectangles.

Rather than having to call a "get rectangle" routine for each possible rectangle, You can use *First_Rect* to get the definition of the first rectangle in the set, then call *Next_Rect* for the next definitions, until you have them all.

The rectangle definition returned is the standard GEM/Pascal library version: coordinates of the upper left corner, *x* and *y*, width, *w*, and height, *h*, all in pixels. If the width and height are *0*, there are no more rectangles in the set.

## RECT_INTERSECT ()

```
b := Rect_Intersect ()

FUNCTION Rect_Intersect ( x1,y1,
                w1,h1 : Short_Integer ;
                VAR x2,y2,
                w2,h2 : Short_Integer )
                : Boolean ;
```

**EXAMPLE:**

```
   ...

Wait_for_Update ( w_handle,mx,my,mw,mh ) ;
Begin_Update ;
First_Rect ( w_handle,rx,ry,rw,rh ) ;

 WHILE rw<>0 AND rh<>0 DO

   BEGIN

    IF Rect_Intersect ( mx,my,mw,mh,rx,ry,rw,rh )
   THEN

   Do_ReDraw ( w_handle,rx,ry,rw,rh ) ;
   Next_Rect ( w_handle,rx,ry,rw,rh ) ;

   END ;

End_Update ;
  ...
```

*Rect_Intersect* takes the standard GEM definitions of a pair of rectangles, pixel coordinates of the upper left corner, width, and height, and finds the rectangle which represents the intersection of the pair. Note that it returns the intersection by modifying the variables passed to define the second rectangle

*Rect_Intersect* is a boolean function: it returns *True* if the rectangles intersect, and *False* otherwise.

# MESSAGES FROM GEM

An application program can receive messages caused by actions taken by the user or the program itself. Messages from GEM are really the indirect result of some user or application program activity. For example, GEM asks for a redraw when a desk accessory is terminated, but the desk accessory only told GEM it was done because the user asked it to.

Personal Pascal implements the messages which GEM has predefined. We've discussed redraw messages, but they aren't the only messages an application program can get. Here is a quick summary of the standard GEM/Pascal messages:

**--Menu item selected**
**--Redraw needed**
**--New window must be moved to front**
**--User clicked on the close box**
**--User clicked on the full box**
**--User clicked on scroll bar arrows**
**--User clicked on scroll bar**
**--User requested scroll slider movement**
**--User requested a window size change**
**--User requested a window movement**

A Personal Pascal program using the GEM/Pascal library finds out about events by calling *Get_Event,* and GEM messages are just one kind of event. Note, also, that an application program can tell *Get_Event* that it only wants notification about particular kinds of events.

In general, though, a program which allows all GEM features, including a menu bar and desk accessories, must always ask for messages when it calls *Get_Event.* Otherwise important information can be lost, especially redraw messages. In some cases you can avoid messages, as when *Begin_Mouse* causes mouse events outside the top window to be ignored. With that in mind, let's see how to determine that a message has been passed, and what it means.

## THE MESSAGE BUFFER

If you examine the contents of the *GEMTYPE.PAS* file, you will find that *Message_Buffer* has been predefined as an array of 16 integers, indexed 0..15. For this and subsequent discussions within this section, assume the declaration:

```
msg : Message_Buffer ;
```

In the following examples, replace *msg* with your own buffer name, if you like. Notice that the first three elements of the buffer always have the same definitions:

**msg [0]** is the message type designator.

**msg [1]** is the application identifier of the sender of the message. Not meaningful under standard GEM with standard desk accessories.

**msg [2]** is non-zero only if the message is longer than the 16 words of this buffer. Again, this is not meaningful with standard GEM messages.

**msg [3]** is always the handle of the window affected by any window-related message.

The meanings of the rest of the buffer elements depend on the message being processed. For window-related messages, *msg [3]* is always a window handle, but depending on the message, it may or may not be the handle of the window currently in front. In any case, a useful skeleton for processing *Get_Event* messages is:

```
CASE msg[0] OF

  MN_Selected : Do_Menu_Selection ;
  WM_Redraw : Do_Redraw ;
  WM_Topped : Do_New_Top_Window ;
  WM_Closed : Do_Close_Box ;
  WM_Full   : Do_Full_Box ;

  ... { et cetera }
```

Because of its nature, the *Message_Buffer* variable needs to be visible to several routines. It's OK to define it as a global variable, since only one such buffer is necessary.

The following explanations describe the purpose of each standard GEM message and the meaning of the *Message_Buffer* parameters for that message. The explanation headings are the names of the messages, as given in the file *GEMSUBS.PAS*.

**MN_Selected**

Purpose:

GEM passes this message when the user clicks on an item in the menu.

**NOTE:** The application program does NOT receive this message when a desk accessory is selected.

Message Buffer Contents:

**msg[3]** is the index of the menu *title* which was selected during the process of clicking on the specific item.

**msg[4]** is the index of the particular menu *item* selected.

These indices refer to the index values returned by *Add_MTitle* and *Add_MItem* when the menu was built. Clicking a de-selected item does nothing.

Usually, your program will have built an array, defining the meaning of each menu item as it was added by *Add_MItem*. In that case, the item index passed here can be compared to the contents of the array, and the appropriate action taken.

Your application program needs the menu title index because it must make a call to *Menu_Normal* or the title will stay highlighted.

**Special case:** If the menu title index is three, i.e.: *msg[3]=3* is *true*, then the user has selected the *info* item under the *Desk* title. The item index will be valid, but unimportant.

## WM_Redraw

Purpose:

Anytime an application program needs to update a window, GEM generates a redraw message.

Message Buffer Contents:

**msg [3]** is the handle of the window which needs to be redrawn.

**msg [4]** is the horizontal pixel coordinate of the upper left corner of the redraw area.

**msg[5]** is the vertical pixel coordinate of the upper left corner of the redraw area.

**msg [6]** is the width of the redraw area in pixel units.

**msg [7]** is the height of the redraw area in pixel units.

**SEE:** *First_Rect; Next_Rect,* page 7-162 and *Rect_Intersect,* page 7-164.

## WM_Topped

Purpose:

Tells the application program that the user has asked that a window be moved in front of the current window, usually by clicking on a window other than the top one.

Message Buffer Contents:

**msg [3]** is the handle of the window which the user wants in front.

If the application program allows the user request, it should make a call to *Bring_To_Front*. Note that this call will cause GEM to issue one or more redraw messages.

Unfortunately, GEM programs do *not* receive this message when other programs want to put one of their windows on the screen.

**WM_Closed**

Purpose:

Tells the application program that the user wants to close the current front window, usually by clicking on the window's close box.

Message Buffer Contents:

**msg [3]** is the handle of the window which the user wants to close. It should be the front window.

If the application program agrees with the user that the window can be closed, it can make a call to *Close_Window.* This call may cause GEM to issue one or more redraw messages.

## WM_Fulled

Purpose:

Tells the application program that the user wants to expand the current front window to its largest possible size or, if the window is already at its largest possible size, tells the program to restore it to its last previous size.

Message Buffer Contents:

**msg [3]** is the handle of the window which the user wants to resize. It should be the front window.

If the application program agrees that the window size can be changed, it can make a call to *Set_WSize.* This call may cause GEM to issue one or more redraw messages.

Usually, the easiest way to implement this call is to remember the maximum size specified via New_Window and the last size given to Set_WSize. There are other ways to obtain these values, using GEM/Pascal library calls still to be introduced, such as Wind_Get.

## WM_Arrowed

Purpose:

This message indicates one of several possible actions within the scroll bar in the current front window. A word in the message buffer tells the application program exactly which action initiated the request.

Message Buffer Contents:

**msg [3]** is the handle of the window which owns the scroll bars. It should be the front window.

**msg [4]** is an *action index* which further identifies the cause of the message. The following table equates an action index to its associated scroll bar area:

| Index | Scroll Bar Area |
|---|---|
| 0 | page up |
| 1 | page down |
| 2 | row up (up arrow) |
| 3 | row down (down arrow) |
| 4 | page left |
| 5 | page right |
| 6 | column left (left arrow) |
| 7 | column right (right arrow) |

If an application program's window displays only part of the available display data, the user might want to *scroll* the displayed area over the rest of the data. For example, if the window shows only ten lines of a 50 line text file, the user may want to scroll up or down within the file, to see the rest of the file.

**NOTE**: By GEM convention, clicking on the various scroll arrows is a request for the smallest possible scrolling movement in the direction specified. With text, this is usually a character row or column, hence the names.

Similarly, the convention is that clicking on the shaded portion of the scroll bar requests a larger movement, such as a *page.* When an unshaded slider area is displayed, clicking outside of it, (but still within the bar) requests a scroll in the direction *away* from the slider and *toward* the click point.

## WM_HSlid and WM_VSlid

Purpose:

These messages are produced when the user clicks on the slider in the appropriate scroll bar and moves it to a new position. They generally request the application program to show an entirely new area of the total possible display.

Message Buffer Contents:

**msg [3]** is the handle of the window which owns the scroll bars. It should be the front window.

**msg [4]** is a number from 0 to 1000 which tells the application program where the slider was moved. If the message is *WM_HSlid*, then zero indicates the leftmost position and 1000 indicates the rightmost. If the message is *WM_VSlid,* then zero is the top of the scroll bar and 1000 is the bottom.

The numbers given by msg[4] may be thought of as being ten times a percentage. Thus, if a text file had 200 lines and the user moved the vertical slider to position 400, you might reasonably start the window display at line 80 (40.0% of 200).

**NOTE:** Although the user has dragged the outline box of the slider to the location specified in *msg [4]*, the slider itself does not move until the application program tells it to. To move the slider, call *Wind_Set* which is documented in the section on Miscellaneous Routines.

As with all scroll bar events, the application program may choose to ignore this message.

## WM_Sized and WM_Moved

Purpose:

These messages are produced when the user requests a change in a window's size or position, using the mouse on the size box or drag bar.

Message Buffer Contents:

**msg [3]** is the handle of the window to be resized or moved. It should be the front window.

**msg [4]** is the horizontal pixel coordinate of the upper left corner of the requested area.

**msg [5]** is the vertical pixel coordinate of the upper left corner of the requested area.

**msg [6]** is the width of the requested area in pixel units.

**msg [7]** is the height of the requested area in pixel units.

If the application agrees with the user's request, it calls *Set_WSize.* On the other hand, if the program doesn't like the request, it can ignore it or modify it. For instance, a program might limit window size.

**NOTE:** For each of the messages, only one pair of numbers in the rectangle definition will change. For *WM_Moved*, the *width and height* should stay the same. For *WM_Sized*, the coordinates of the upper left corner should not change. The user may *not* request both changes at once.

BLANK
PAGE

# MISCELLANEOUS ROUTINES

This section describes GEM-related routines which have multiple purposes or which have no direct connection to one of the topics already covered.

**WIND_GET ()**
**WIND_SET ()**

```
PROCEDURE Wind_Get ( handle,
                     request : Short_Integer ;
                     VAR v1,v2,
                     v3,v4 : Short_Integer ) ;

PROCEDURE Wind_Set ( handle,
                     request,
                     v1,v2,
                     v3,v4 : Short_Integer ) ;
```

**EXAMPLES:**

```
Wind_Get ( cur_window, WF_FullXYWH, x,y,w,h ) ;

Wind_Set ( cur_window, WF_VSlide, msg[4],0,0,0 ) ;
```

This matched pair of routines is the Personal Pascal implementation of GEM's catch-all window information routines. Many of the routines in the GEM/Pascal library are implemented by judicious use of calls to one of these routines. For example, both *Work_Rect* and *Border_Rect* consist of nothing more than calls to *Wind_Get* with the appropriate request word.

There are a few uses of these routines that occur so infrequently that they don't justify custom procedures. Instead, you can use these general calls to implement more specific routines you might need.

Using these procedures, your program can get information about window parameters using *Wind_Get*, or change the state of the window with *Wind_Set*. The difference between *get* and *set* is whether the parameters define changes ( *set* ) or request current values ( *get* ).

The first parameter to either routine is always a window handle previously obtained by *New_Window*. Note, however, that a window handle of *zero* is a special case which indicates the entire desktop. The zero window handle is only valid for some requests.

The second parameter is a request name, actually a number as described below. The third through sixth parameters are integers which *define* new values for *Wind_Set* or receive *current* values from *Wind_Get*.

## WIND_GET, WIND_SET ( Continued )

**NOTE:** *Wind_Get* requires the use of variables in these positions, since they will be modified when the routine is called.

**ALSO:** If a parameter isn't used for a particular request, it is not mentioned, but the procedure call must still pass all parameters, even if some are dummies.

For each request name, the meanings of the value parameters ( *v1, v2, v3, v4* ) are given on the following pages.

*WIND_GET, WIND_SET ( Continued )*

**Wind_Get ( WF_PrevXYWH,v1,v2,v3,v4 )** requests the standard GEM definition of the rectangle which defines the last previous size of the given window. This request name is only valid for *Wind_Get.*

**v1** is given the horizontal pixel coordinate of the upper left corner of the designated window's previous *position.*

**v2** is given the vertical pixel coordinate of the upper left corner of the designated window's previous *position.*

**v3** is given the *width*, in *pixels*, of the designated window's previous *size.*

**v4** is given the *height,* in *pixels*, of the designated window's *previous size.*

*WIND_GET, WIND_SET ( Continued )*

**Wind_Get ( WF_FullXYWH, v1,v 2, v3, v4 )**
requests the standard GEM definition of the rectangle which defines the largest possible size of the given window, as passed by the application program when *New_Window* was called. This request name is only valid for *Wind_Get.*

**v1, v2, v3, v4** are as given for *WF_PrevXYWH* except that they represent the *x,y,w,h* values passed to *New_Window.*

**Wind_Get ( WF_HSlide, v1, v2, v3, v4 )**
**Wind_Set ( WF_HSlide, v1, v2, v3, v4 )**
requests a change to ( *Wind_Set* ) or the current value of ( *Wind_Get* ) the relative position of the *horizontal* slider within the scroll bar.

**v1** is a number from 1 to 1000 representing relative position. A value of *1* indicates the *leftmost* position. A value of *1000* indicates the *rightmost* position.

**Wind_Get ( WF_VSlide, v1, v2, v3, v4 )**
**Wind_Set ( WF_VSlide, v1, v2, v3, v4 )**
requests a change to ( *Wind_Set* ) or the current value of ( *Wind_Get* ) the relative position of the *vertical* slider within the scroll bar.

**v1** is a number from 1 to 1000 representing relative position. A value of *1* indicates the *topmost* position. A value of *1000* indicates the *bottom-most* position.
**NOTE:** These duplicate *WF_HSlide* and *WF_VSlide.*

*WIND_GET, WIND_SET ( Continued )*

Wind_Get ( WF_HSlSize, v1, v2, v3, v4 )
Wind_Set ( WF_HSlSize, v1, v2, v3, v4 )
requests a change to ( *Wind_Set* ) or the current value of ( *Wind_Get* ) the relative size of the *horizontal* slider within the scroll bar.

**v1** is a number either from 1 to 1000, or -1, which represents the relative size. The number may be thought of as a percentage times ten, thus producing values from 0.1% to 100%. A value of -1 indicates the default slider size: a square box.

**Wind_Get ( WF_VSlSize, v1, v2, v3, v4 )**
**Wind_Set ( WF_HSlSize, v1, v2, v3, v4 )**
requests a change to ( *Wind_Set* ) or the current value of ( *Wind_Get* ) the relative size of the *vertical* slider within the scroll bar.

**v1** the meaning and use of *v1* is the same as for *WF_HSlSize.*

*WIND_GET, WIND_SET ( Continued )*

As an example, the following listing implements a call that simplifies a slider position request:

```
FUNCTION v_slide_position (handle : Short_Integer)
                : Short_Integer ;

  VAR

    v1,v2,v3,v4 : Short_Integer ;

  BEGIN

    Wind_Get ( handle, WF_VSlide, v1, v2, v3, v4 );
    v_slide_position := v1 ;

  END
```

BLANK
PAGE

## SYS_FONT_SIZE ()

```
PROCEDURE Sys_Font_Size ( VAR cw,ch,
                          bw,bh : Short_Integer ) ;
```

Use Sys_Font_Size to find the width and height, in pixels, of characters in the current screen resolution. The first two variables, *cw* and *ch*, receive this information. You might, for example, divide the width of the current *Work_Rect* by *ch* to determine the width of a window in characters.

The second two variables, *bw* and *bh,* receive the width and height, again in pixel units, of a square box large enough to hold a single character. GEM uses this information internally, for example, when working with the arrows in a scroll bar. No GEM/Pascal library routine uses these two parameters.

## CLEAR_SCREEN

PROCEDURE Clear_Screen ;

Use Clear_Screen with caution; it unconditionally clears the *entire screen*, windows, dialog boxes, menus and all.

Clear_Screen should generally be used by programs which use the entire screen and do not use a menu bar. You *can* erase the menu bar, clear the screen, and then redraw the menu bar, but a better way would be to use *Paint_Rect* to clear the area defined by *Work_Rect* of window 0.

## SAVE_SCRN ()
## RESTR_SCRN ()

PROCEDURE Save_Scrn( VAR buf : Screen_Type )

PROCEDURE Restr_Scrn( VAR buf : Screen_Type )

These procedures allow you to save a screen to a buffer or to restore a screen from a buffer. The Screen_Type buffer is a record in *D.E.G.A.S.* format. Good for:

Cleaning up a screen after an information box writes all over it.

Saving screen sequences for animations, environment switching, etc.

Restoring screens for the same reasons.

**SEE:** The *GEMTYPE.PAS* file for the definition of the *D.E.G.A.S.* record format.

## READ_SCRN ()
## WRITE_SCRN ()

```
i := Read_Scrn ()
i := Write_Scrn ()

FUNCTION Read_Scrn( File: String;
                    Buf: Screen_Type )
                    : Err_Code

FUNCTION Write_Scrn( File: String;
                     Buf: Screen_Type )
                     : Err_Code
```

These functions read a screen from a disk file or write a screen to a file. The *Screen_Type* buffer is in *D.E.G.A.S.* format.
**SEE:** The GEMTYPE.PAS file for the definition of the *D.E.G.A.S.* record format.

## LOAD_RESOURCE ()

```
b := Load_Resource ()

FUNCTION Load_Resource ( file : Path_Name )
                : Boolean ;

({ somewhere in the global CONST section,

 the programmer must have a line similar to this: }

  {$I ADVENT.I }
                { ADVENT.I is a resource file name.

                  then these program lines will work:}

IF NOT Load_Resource( 'ADVENT.RSC' ) THEN

BEGIN

   Junk := Do_Alert( '[3][No resource file!][Quit]' );

   Exit_Gem ;

   Halt ;

END
```

Even though the OSS GEM/Pascal library provides the means of building alert boxes, menus, and dialog boxes dynamically, there might be times when you would choose to use Atari's Resource Construction Set, a part of the software developer's package, ( or another compatible RCS ) to construct these GEM building blocks.

One of the options in most resource construction sets allows for the creation of Pascal-compatible files. Two files are created, both with the same initial name but with different extenders. Usually, the file with the *.I* extender is an *ASCII* file which consists of Pascal CONSTant definitions. You can either merge the .I file into a source file using the editor or include it at compile time using the ( {$I} ) directive.

The other file is the *resource* file which contains object trees, and the like. Luckily, the Personal Pascal programmer need not be aware of its exact contents. Simply load it by calling Load_Resource and then use the indices defined in the .I file.

**NOTE:** This function returns a *boolean* value. If the file was found and successfully loaded, it returns *True.* If the load was unsuccessful for any reason, it returns *False.*

**FIND_MENU ()**
**FIND_DIALOG()**
**FIND_ALERT ()**

```
PROCEDURE Find_Menu ( index : Short_Integer ;
                      VAR r_menu : Menu_Ptr ) ;

PROCEDURE Find_Dialog ( index : Short_Integer ;
                      VAR r_dial : Dialog_Ptr ) ;

PROCEDURE Find_Alert ( index : Short_Integer ;
                      VAR r_alert : Str255 ) ;
```

**EXAMPLE:**

```
{ assumes that BIGMENU is defined in the
  CONSTants in the ".I" file and that
  the resource (".RSC") file has been loaded }

VAR

  BigMenu_Ptr : Menu_Ptr ;
  ...
Find_Menu( BIGMENU, BigMenu_Ptr ) ;
Draw_Menu( BigMenu_Ptr );

  ...
```

Once the resource file has been loaded, the application program can access its contents as needed, provided only that it knows the index values for the desired items. If the program has properly included the corresponding *./* file, then it can use the names defined therein.

**NOTE:** The names are those given by the resource builder when the resource file was constructed.

For menus and dialogs, the application program needs a pointer of the proper type to pass along to *Draw_Menu* or *Do_Dialog.* For alert boxes, the program receives a standard Personal Pascal string, of maximum size, which it can pass to *Do_Alert* or use in any other way it chooses.

**NOTE:** Menu title indices, dialog and menu item indices, etc., are given names by the person who built the resource file. These same names appear in the *./* file and can be used with such calls as *Menu_Normal, Obj_SetState,* etc., just as if they were integer indices returned by *Add_MTitle*, *Add_DItem*, etc.

## FREE_RESOURCE

PROCEDURE Free_Resource ;

When a program is completely finished with the contents of a resource file, it can release its space back to the system using Free_Resource. Any and all dialog boxes or menus built into and used from the resource file must be removed from the screen first. Usually, alert box strings are copied to the program's variable space and so are safe.

Free_Resource is seldom useful, but in a few circumstances, especially where a program progresses in stages, it might make sense to get rid of one set of resources and then load another.

# GENERIC CALLS

## MAKING GENERIC BIOS, XBIOS, and GEMDOS CALLS

There are literally dozens of operating system calls available to the TOS programmer. Many of these are translated into Personal Pascal's standard I/O library (e.g., the TOS call "f_open" relates closely to Pascal's "Reset" procedure).

However, we do understand that you may wish to use some of the features that we do not support. There are two choices available to you:

(1) Buy "Tackle Box ST". This package provides the Personal Pascal user with access to virtually every TOS routine. You can order this package directly from the manufacturer or buy it directly from OSS. See the description and ordering information at the end of this manual.

(2) You can buy the book "Atari ST Internals" published by Abacus. As a translation of a German book (from Data Becker, of Dusseldorf, West Germany), there are places where it is somewhat cryptic. But every TOS call is described, and with a little experimentation you should be able to implement every call.

If you choose the second option, then you also need to know how to make a call to the GEMDOS, BIOS, and XBIOS routines described in that book. This appendix describes how to make those calls. You are able to make these calls thanks to Personal Pascal's language directives: GEMDOS, BIOS, and XBIOS.

The Personal Pascal compiler is smart enough to generate the proper code to access these routines, providing you tell it what parameters those routines are expecting. This documentation will provide you with enough information to write Pascal programs that call the BIOS, XBIOS, or GEMDOS.

We do not intend to provide documentation for every one of these calls (that is what "Tackle Box ST" and/or "Atari ST Internals" are for!), but we will give you declarations for some of the more useful routines. We also suggest that you refer to the demo programs available on the OSS BBS (and other places). Several of these implement other TOS calls.

The calls we will describe are divided into two groups: Character-oriented I/O and Disk I/O operations.

## TOS CALLS: CHARACTER I/O

The ST supports five character oriented devices. Of these five, you will probably only use four, the printer port, the RS232 port, the MIDI port, and the keyboard/console. The other device is the internal data path to the intelligent keyboard unit (which handles the mouse and joysticks, as well as the keyboard); you will seldom, if ever, need to access that device directly!

One of the most common operations on character oriented devices is reading a single character. The following few routines perform that function:

### GET CHARACTER FROM CHARACTER-ORIENTED DEVICE.

This routine is the underlying BIOS call which can be used to perform input from any of the five devices:

```
FUNCTION bconin( device : Short_Integer ) : Long_Integer ;
    BIOS( 2 ) ;
```

This function returns a character from the specified character-oriented device. The valid values for the device parameter are as follows:

0 printer port (not used for input)
1 RS232 port
2 keyboard
3 MIDI port
4 intelligent keyboard (don't use!)

If no character was waiting when bconin was called, it waits until a character is available. If you don't want to wait for characters, you should call bconstat first, to determine that a character is available. The bconin function returns the character value in the low byte of the returned Long_Integer.

If the specified device is the console (device 2), however, the return value is more complex. In that case, the keyboard scancode is returned in the upper word, and the ASCII equivalent (if any) is returned in the lower word. If you only want the Short_Integer return value, simply assign it to a Short_Integer variable.

You may wish to find out whether a character is available before calling one of the character input routines.

### CHARACTER-ORIENTED DEVICE INPUT STATUS

```
FUNCTION bcistat( device : Short_Integer ) : Boolean ;
    BIOS( 1 ) ;
```

This function expects the number of a character oriented device, as described above (0-4). It returns a True value if at least one character is waiting for input and False otherwise. (If the device is the printer, however, the returned status may not be correct if the ST is powered up while the printer is off-line.) You might want to define your own special-purpose status routine as follows, so you don't have to insert the device except in one place:

```
(* Return True, if there is a keyboard character waiting. *)

FUNCTION Char_Waiting : Boolean ;

  CONST
   keyboard = 2 ;  (* Device number of the keyboard. *)
  FUNCTION bcistat( device : Short_Integer ) : Boolean ;
       BIOS( 1 ) ;
  BEGIN
   Char_Waiting := bcistat( keyboard ) ;
  END ;
```

Besides character input and input status, you also need to be able to put characters to character devices.

**PUT CHARACTER TO CHARACTER-ORIENTED DEVICE.**

```
PROCEDURE bconout( device, c : Short_Integer ) ;
      BIOS( 3 ) ;
```

This routine writes a single character to the specified device. If the device's output buffer is full, the routine will wait until the character is actually placed in the buffer. If you don't want to wait for output, you should call bcostat first, to determine that the device is ready to receive the next character.

CHARACTER-ORIENTED
DEVICE OUTPUT STATUS

```
FUNCTION bcostat( device : Short_Integer ) : Boolean ;
    BIOS( 8 ) ;
```

This routine checks to see whether the specified device is ready to accept another character. It returns True, if the device is ready to receive, and False otherwise.

**CAUTION**: If the ST is powered up while the printer is off-line, the hardware does not detect the off-line condition. The bcostat call will return True even though the printer is not ready to accept data. As soon as the printer is turned on-line again, the status is correct.

## GEMDOS DISK FILE OPERATIONS

This section describes several of the GEMDOS operations that are available to Personal Pascal via the GEMDOS language directive.

Once your program has declared these subprograms as shown below, you can perform the given operations by simply calling these routines by name, passing appropriate parameters.

In these declarations, the parameter type Path is used often. Path has been declared thus:

```
TYPE
  Path = PACKED ARRAY [ 1..80 ] OF Char ;
```

(The maximum length of a GEMDOS pathname is 80 characters.)

You can use the P_to_CStrg procedure (described in section 7) to convert a Pascal string to a Path variable type.

Create and Open a File

Sometimes, you may want open a file with special properties that Pascal doesn't support. For this purpose, you can use the following routine:

```
FUNCTION f_create( VAR name : Path ;
                   attributes : Short_Integer )
                 : Short_Integer ;
    GEMDOS( $3C ) ;
```

This call creates a new file with the specified name and the specified attributes. Excepting for the ability to specify attributes, this call is functionally the same as Pascal's standard REWRITE procedure. The bits in the attributes parameter have the following assignments:

| BIT | MEANING |
|---|---|
| $01 | file is read-only |
| $02 | file is hidden from directory search |
| $04 | file is a system file, hidden from directory search |
| $08 | file contains a volume label in the first 8 data bytes |

The return value is a valid GEMDOS file handle, if greater than or equal to zero, or an error number, if negative. You should use this call to open a file for output, if you want to open a new file, or if you want to first erase the previous contents. If you want to write to an existing file, without erasing the contents, use the f_open call, below.

### Open a File

You might also want to open an existing file (or one you created with f_create) without using the built-in procedure RESET. You can use this GEMDOS call:

```
FUNCTION f_open( VAR name : Path ;
                 mode : Short_Integer )
                 : Short_Integer ;
    GEMDOS( $3D ) ;
```

Use this call to open a file for reading, writing, or updating. If you want to open a file for writing, but you want to first erase the previous contents, use the f_create call, instead. The valid values for mode are:

0 open for reading only
1 open for writing only
2 open for reading or writing

The return value is a GEMDOS handle, if greater than or equal to zero, or an error number, if negative. Notice that this call does not have a parameter to specify the attributes of the file. Those attributes are set by the f_create call and are not changed by this call. If you want to change the attributes of a file, you can use the f_attrib call, below.

**Close an Open File**

If you used f_create or f_open to ready a file for access, you should use the following call to close it when you're finished reading or writing to the file:

```
FUNCTION f_close( handle : Short_Integer )
            : Short_Integer ;
    GEMDOS( $3E ) ;
```

The parameter handle should be the same as that returned by the appropriate open call. Zero is returned, if the file was closed successfully, or a negative error number, otherwise.

**Read Bytes from a File**

Pascal supports reading from and writing to files one item at a time, where the size of the item is the size of the file pointer variable. Occasionally you may want to read or write in larger chunks, especially if your item size is small, since GEMDOS isn't very fast for single-byte transfers. Or perhaps you may wish to read varying kinds of information from a file. The following call allows you to read a block of characters into memory:

```
FUNCTION f_read( handle : Short_Integer ;
                 count : Long_Integer ;
                 VAR buf :  Buf_Type )
                : Long_Integer ;
                 GEMDOS( $3F ) ;
```

This call reads an arbitrary number of bytes from a file into a desired buffer. The number of bytes actually read is returned, if the function was successful, or a negative error number, if something went wrong. Note that the number of bytes actually read may be shorter than the number of bytes requested, if the end-of-file position was reached. The Buf_Type mentioned above may be almost any type, but each type will require a separate definition of this call (with a different function name). For example, to read 100 two-byte values into an array, you might use a program segment like this:

```
TYPE
  Hundred_Integers = ARRAY [ 1..100 ] OF Short_Integer ;

VAR
  a : Hundred_Integers ;
  bytes_read : Long_Integer ;

PROCEDURE read100 ( handle  :  Short_Integer  ;
                    count  : Long_Integer ;
                    VAR buf : Hundred_Integers )
                    : Long_Integer ;
       GEMDOS( $3f ) ;

BEGIN
  bytes_read := f_read( handle, 200, a ) ;
END ;
```

Note that 200 was passed as the number of bytes to read, since we wanted 100 two-byte values!

The handle parameter should be that value returned by either the f_create or f_open call. If you want to use the f_read call to read from a file which was opened using the built-in procedure Reset, you can use the built-in function Handle to find out the handle associated with the file. If you are reading from a file just opened using Reset, you must be aware, however, that the first item has already been read from the file and put into the file buffer variable.

**Write Bytes to a File**

Similarly, you may want to write an arbitrary number of bytes to a file. The following call supports block writing:

```
FUNCTION f_write( handle : Short_Integer ;
                  count : Long_Integer ;
                  VAR buf : Buf_Type )
                 : Long_Integer ;
                  GEMDOS( $40 ) ;
```

This call is the counterpart of the f_read function described above. It takes an arbitrary number of bytes from a buffer and outputs them to a previously opened file. The handle parameter must be that which was returned by a previous f_open or f_create call. You can also use the Handle function to get the handle of a file which was opened using the Rewrite built-in procedure. The value returned by f_write is the number of bytes written, if the operation was successful, or a negative error number. In general, if the number of bytes returned does not equal the number requested, something went wrong!
As with f_read, you must declare this function as many times as you have different buffer types to be written.

### Delete a File

There is no standard procedure in Pascal to remove a file from a disk, so if you want to erase files, you need the following call:

```
FUNCTION f_delete( VAR name : Path )
                : Short_Integer ;
                 GEMDOS( $41 ) ;
```

Zero is returned if the delete was successful. A negative error value is returned otherwise.

### Seek Within a File

Personal Pascal supports random access to files using the built-in procedures Get, Put, and Seek. If you want to use instead the underlying GEMDOS routine to position within a file, here it is:

```
FUNCTION f_seek( offset : Long_Integer  ;
                 handle : Short_Integer )
                 mode : Short_Integer )
                : Long_Integer ;
                 GEMDOS( $42 ) ;
```

Use this call to point to a particular byte position within a file. The offset parameter specifies the desired byte position, and the mode parameter specifies which file position the offset parameter is relative to:

MODE RELATIVE TO

0 the beginning of the file
1 the current location
2 the end of the file

The offset parameter is signed, so you could, for example, move 10 bytes backwards in the file by specifying offset and mode parameters of -10 and 1, respectively. You can append to a file by specifying an offset of zero with a mode of 2 (end of file).

### GET/SET FILE ATTRIBUTES

As mentioned above, the f_create call sets a file's attributes. These attributes are never changed when the file is subsequently opened. If you ever want to change the attributes of a file, you should use the following call:

```
FUNCTION f_attrib( VAR name : Path ;
                   mode : Short_Integer ;
                   attributes : Short_Integer )
                 : Short_Integer ;
                   GEMDOS( $43 ) ;
```

The mode parameter specifies whether to get the file attributes, if 0, or to set the attributes, if 1. The attributes parameter is specified in the same way as for the f_create call, above, with the last two being additions:

BIT MEANING

$01 file is read-only
$02 file is hidden from directory search
$04 file is a system file, hidden from directory search
$08 file contains a volume label in the first 8 data bytes
$10 file is a subdirectory
$20 file is written and closed correctly.

The last two attributes refer only to subdirectories.

# PORT CONFIGURATION

If you are writing a program which performs I/O to one of the devices that connect to the back of the ST (i.e., a printer or a modem), you may wish to set the configuration in your program rather than relying on a desk accessories to allow the user to configure the ports. If you want to set the configuration of the RS232 port or the parallel port, you need to know a few calls:

**SET THE PRINTER CONFIGURATION.**

The following XBIOS call allows you to configure the printer:

```
FUNCTION setprt( config : Short_Integer ) : Short_Integer ;
      XBIOS( 33 ) ;
```

In order to set or get the current printer configuration, you should use this call. If the config parameter is passed as -1, the current configuration is passed back as the return value. Otherwise, config specifies the desired configuration of the printer. The various bits within config specify the configuration as follows:

| bit# | when 0 | when 1 |
|---|---|---|
| 0 | dot matrix | daisy wheel |
| 1 | color printer | monochrome |
| 2 | Atari printer | Epson compatible |
| 3 | draft mode | final mode |
| 4 | parallel port | RS232 port |
| 5 | continuous paper | single sheet |
| 6 | reserved | |
| 7 | reserved | |
| 8 | reserved | |
| 9 | reserved | |
| 10 | reserved | |
| 11 | reserved | |
| 12 | reserved | |
| 13 | reserved | |
| 14 | reserved | |
| 15 | MUST BE ZERO! | |

## CONFIGURE THE RS232 PORT.

The following XBIOS call sets the various parameters controlling the RS232 port:

```
PROCDURE rsconf( speed, flowctl, ucr, rsr, tsr, scr :
Short_Integer ) ;
  XBIOS( 15 ) ;
```

If any of the parameters is -1, the corresponding RS232 parameter is left unchanged from its previous value. You will mostly be dealing with setting the baud rate, which is governed by the speed parameter:

| SPEED | RATE |
|---|---|
| 0 | 19200 |
| 1 | 9600 |
| 2 | 4800 |
| 3 | 3600 |
| 4 | 2400 |
| 5 | 2000 |
| 6 | 1800 |
| 7 | 1200 |
| 8 | 600 |
| 9 | 300 |
| 10 | 200 |
| 11 | 150 |
| 12 | 134 |
| 13 | 110 |
| 14 | 75 |
| 15 | 50- |

The last value, 15, may not generate an accurate (as if you'll ever need it!). You may also need to change the flow-control option of the RS232 port. It is specified in the flow parameter as follows:

FLOW FLOW-CONTROL

0 No flow control
1 XON/XOFF (control-S/control-Q)
2 RTS/CTS
3 XON/XOFF and RTS/CTS

The value 3 doesn't represent a very useful condition, but it should work. The other four parameters set registers within the 68901 chip (for a more complete, but still sketchy, discussion, see the book "Atari ST Internals"). These registers perform the following functions:

REGISTER FUNCTION

ucr USART control register
rsr Receiver status register
tsr Transmitter status register
scr Synchronous character register

If you are transmitting in asynchronous mode (i.e, almost always), you will probably only use the ucr parameter, which has the following meanings:

UCR BITS FUNCTION

0 unused
1 parity type: 0=odd 1=even
2 parity enable: 0=no parity 1=parity
4,3 0,0 -> synchronous mode (all others asynch)
0,1 -> 1 start bit, 1 stop bit
1,1 -> 1 start bit, 2 stop bits
6,5 number of data bits
0,0 -> 8 bits
0,1 -> 7
1,0 -> 6
1,1 -> 5
7 transmit and receive frequency
0 -> divide by 1 (synchronous only)
1 -> divide by 16

# USING GET_IN_FILE

No standard Atari manual describes how to use the Get_In_File "item selector". The following is a short description of this process.

Generally, the item selector is self explanatory: The user may double click on any file name displayed in the scrolling window. If the file he or she wants is not visible in that window, the scroll bars may be used to display other portions of the list of files in the directory named in the line above the list. Alternatively, the user may simply type in the name of the desired file, so long as the text cursor is placed in the name area in the right side of the dialog box.

If the directory name displayed is not the desired one, the user may select a subordinate directory by clicking on its name in the scrolling window. Or, if the desired directory is the parent (or sibling) of the current directory, the user may click on the close box to go "up" a level in the directory tree.

However, if the user wishes to change either the drive specifier or the wild card search path (displayed at the end of the directory specifier AND at the top of the scrolling window), then he/she must use the mouse (with a click) to move the text cursor to the directory line. Then, after using cursor keys, etc., to edit the search path, the user MUST NOT HIT THE RETURN KEY. Doing so is treated as an escape (cancel) from the dialog.

Instead, the mouse must be moved to point the mouse arrow into the scroll bar of the list window and then the mouse button must be clicked once. The directory displayed will be changed to the user's choice and then a file within that directory may be chosen, as described above.

# MODULAR COMPILATION

One of the failings of standard Pascal is its lack of the ability to break a large program into smaller units which can be compiled separately. Personal Pascal solves this deficiency by providing a rudimentary yet powerful method of performing "modular compilation." In this section, we will provide a simple example of using modules, as well as some guidelines and hints. First, the example:

Consider the following simple program:

```
1 PROGRAM simple;
2   VAR i: integer;
3   BEGIN
4     FOR i := 1 TO 10 DO writeln( i );
5   END.
```

Just for the purposes of this example, lets say we want to call a routine "print_message" instead of "writeln" in line 4. We also want to put that routine into a different file so we can compile them separately. We need to create two files, one of which will be our "module." First, here is the "main file", which we will assume is called EXAMPLE.PAS:

```
PROGRAM main_file;
  VAR i: integer;

 PROCEDURE print_message( n: integer );
    EXTERNAL;
  { Then the main routine is just like before: }
 BEGIN
    FOR i := 1 TO 10 DO
       print_message( i );
      { Call print_message, not of WriteLn!! }
 END.
```

## MODULAR COMPILATION ( Continued )

Before we go on to the module, lets look at a few things:

(1) The modular compilation flag (M+) didn't appear anywhere in this file! Why? Because you use the M+ flag in each "module" file EXCEPT the one holding your main routine. Otherwise, you'll get link errors.

(2) We declared print_message just as we would have if we were going to code it in this file, but instead of the body of the procedure, we just have the directive EXTERNAL. Now we want to compile this main file to produce a file EXAMPLE.O, the "object" file. But first, we must turn OFF the "Chain to linker" flag in the compiler options dialog box. (Also, we set the compiler to compile for TOS, since we're just going to be using "writeln" to print to the screen.) Assuming that EXAMPLE compiled successfully, lets move on to the "module" file, which we'll call MODULE.PAS:

```
{$M+,E+}
{ This is a module--
  we want its procedures to be visible }
PROGRAM module;

  PROCEDURE print_message( n: integer );
    BEGIN
      writeln( 'In the module with parameter ', n );
    END;

  BEGIN
  { This main routine MUST be empty! }
  END.
```

If you type this in and compile it (again with "Chain to linker" OFF!), you will get a file MODULE.O. Now we want to link both EXAMPLE.O and MODULE.O together with the Pascal libraries to produce a final program file. Put the name "module.o" in the "Additional Link Files" field of the linker options dialog box. Then choose the linker from the File menu, and select the file EXAMPLE.O. The linker will first go to "example.o", then "module.o", then the libraries, in order to produce a final object file EXAMPLE.TOS, which you can run to see the results of our simple example.

### A NOTE ON GLOBAL VARIABLES

In our sample module, we did not declare any global variables. If we wanted to access the global variables that were declared in the main program (just the integer i, in this case), we would have had to declare ALL the global variables THE SAME WAY AND IN THE SAME ORDER AS THE MAIN PROGRAM. In order to make this simpler, put all your global declarations into a file, then use the include ( {$I file } ) directive to insert these into all your files (the main routine, too).

## COMMENTARY

(1) As you can tell, our example did not demonstrate any advantage of using modular compilation. In fact, we went to more work that we would have by having just one source file! In general, if your program if fairly small, you will not benefit from breaking your program up. On the other hand, if your program is quite large, you can save a lot of compile time by splitting it up into several parts. If possible, you should form the modules so that routines with similar functions are in the same module. The Personal Pascal compiler was generated in this way. It is formed of several modules. When you use modular compilation, keep the following points in mind:

Be sure to turn OFF "Chain to linker"

Use the M+,E+ directives ONLY in modules, NOT in your main program

The main program segment in a module MUST be emtpy:

```
BEGIN
END.
```

If you want to access any global variables from modules, all global VAR declarations must also be in the module. We suggest putting your global CONST, TYPE, and VAR declarations into a separate file, and just include it in all modules AND in your main program.

(2) Version 2's COMPILE ALL manager option was designed for modular compilation. After making any changes to the global declarations file--the one that should be included in each module--you could choose COMPILE ALL, presuming that you have previously set up the list of compile files. This is the safest way to make changes safely in all modules.

# Using Personal Pascal with a CLI

Several Command Line Interpreters are now available for the ST, including (for example only) Micro C-Shell. And Personal Pascal was designed from the start to work in such an environment.

If you prefer, you can ignore the PASCAL.PRG "shell" and use COMPILER.PRG, EDITOR.PRG, and LINKER.PRG directly. Each of these programs has some options, summarized here:

**EDITOR fnm [ line [ col [ err ] ] ]**

You MUST pass a file name ("fnm") to the editor. You may optionally pass a line number, in which case when the editor loads and runs the cursor will be positioned at the specified line number. (CAUTION: be sure to use a line number that exists in the file!)

If you pass a line number, you may also pass a column number. Again, if you do so, the cursor will be placed at that column in the given line.

Finally, only if you specify a line number and a column number, you may also specify an error number. If you do so, and if that error number is the same as one in the file named "ERRORS.TXT" then the corresponding error message will appear at the top of the editor screen.

Example:

EDITOR MYPROG.PAS 20 10 167

Causes the Editor to load the file MYPROG.PAS, set the cursor at line 20 and column 10, and displays the error message "Undeclared Label" at the top of the screen.

**COMPILER fnm [/GEM] [/NOCODE] [/ACC] [/DEBUG] [/NOCHECK] [/CHECK] [/CLEAR]**

You MUST pass a file name ("fnm") to the compiler. The source file MUST have the extension ".PAS" (as in MYPROG.PAS). The object file produced by the compiler WILL be given a name of the same form as the source file (in the same directory) but with the extension ".O" (e.g., MYPROG.O).

You may pass one of several optional parameters. Each must be preceded by a slash ("/") and they must be separated from each other by a space. All the options are INDEPENDENT of each other, and each overrides whatever the compiler's default condition is!

Each option is explained below:

**/GEM**

Compile for GEM. Default is compile for TOS. Roughly same as coding the compiler directive $U10 (reserve 10kb or memory for GEM, in other words).

**/NOCODE**

Do not produce an object file! This is really handy when you are doing the first few compiles of a program, when you're expecting several errors. Obviously, the default is DO produce an object file.

**/ACC**

Make this program a desk accessory. Default is a GEM or TOS program.
his has the same effect as coding the compiler directive "$A+" in your program.

**/DEBUG**

Causes compiler to generate the code needed for the "debug" mode (see PP manual). Same as compiler directive "$D+".

**/NOCHECK**

Same as using the compiler directive "$T-". That is, it turns of stack/heap overlap checking. This checking is NOT a lot of overhead, so we do not recommend using this option until your code if pretty well finalized.

**/CHECK**

Same as using BOTH of the compiler directives "$P+" and "$R+".

**NOTE:** This directive is definitely NOT the opposite of /NOCHECK. The two are completely separate and independent!

Pointer and range checking DO use considerable system overhead, but we still recommend that you use them until you are reasonably sure that your program is not pointing off into never-never land.

**/CLEAR**

Same as compiler directive "$C+".

Clearing variables to zero is NOT a thing that most Pascal compilers do, so for compatibility you should not use this directive. Exception: if you are converting from Turbo Pascal, this might be handy.

**EXAMPLE:**

**COMPILER MYPROG /GEM /CHECK /DEBUG**

**LINKER fnm1,fnm2,fnm3,...fnmN**
or
**LINKER fnm.ext=fnm1,fnm2,fnm3...,fnmN**

The LINKER has no specifiable options but it does have two ways to specify the files to be linked and the name of the resultant output file.

The first form shown above will cause all the specified files to be linked together. The files may be either object (".O") files or library files (such as PASLIB and PASGEM). The output file will ALWAYS be named "fnm1.PRG". That is, it will have the same name as the first file in the list but will have the extension ".PRG". If the program is not a GEM program, we recommend renaming the file after it is produced.

The second form shown allows you to specify ANY file name for the output file. This is handy when you want the output file to be in some other directory or to have some extension other than ".PRG". (For example, the PASCAL GEM-based shell specifies ".TOS" when you click on the "link for TOS" button.)

**REMEMBER:** PASLIB should probably be the last file linked in any case. If you are linking for GEM, then PASGEM should be the next to last file in the list.

**BUG:** Although you may specify almost any pathname for each of the given
iles, you can NOT use a name that starts with "..\". If you need to go up a level in the directory, you will have to do so by giving the path more explicitly.

# CREATING DESK ACCESSORIES

There are two ways to cause Personal Pascal version 2 to treat your program as a desk accessory. First, you can use the compiler directive, {$A+}, as described in the section on Compiler Directives. Second, you can simply click on the "ACC" boxes in the compiler and linker options dialogs (again, from the manager).

Using either directive tells the compiler to generate a desk accessory rather than a stand-alone application. However, there are a few more things which you must do to create a successful accessory.

(1) You need to use one or two additional compiler directives: You MUST turn debug mode off via {$D-}. You might also need to specify a stack size via {$Snn} (where the "nn" is a number representing the number of kilobytes of stack your routine needs). A stack size of 5K bytes is given to your accessory by default, and that works for many accessories, but those that (for example) build large dynamic dialogs, use large local arrays, or use many NEW pointer variables may need a bigger stack. If so, try 10K first and then successively larger sizes.

In other words, a common usage might be {$A+,D-,S10} in order to tell the compiler to generate a desk accessory, turn debug mode off, and set the stack size to 10K.

(2) The value actually returned by the Init_Gem routine is the application identification number for your program. Application programs may ignore this value since only one such program may be active in GEM. Because multiple desk accessories may be active, though, your accessory program must save this number in a (global) variable, instead of just testing it. Look at the sample accessory (ACCDEMO.PAS) we have supplied on the disk for details.

(3) Two additional GEM messages are sent to accessories. These messages are AC_Open (40 decimal) and AC_Close (41 decimal). Their meanings are fairly obvious. You MUST respond to these messages as shown in the sample accessory.

(4) PASGEM and GEMSUBS.PAS have one special function declared, specifically for use by desk accessories:

```
FUNCTION Menu_Register( id : Short_Integer ;
                        VAR name : Str255 )
              : Short_Integer ;
                EXTERNAL ;
```

You use the Menu_Register function to insert the name of YOUR accessory into the "Desk" menu! Note that the name string should be a global string! Again, see the sample accessory for details.

After you have compiled and linked your desk accessory, copy it to a BACKUP of your boot disk, and reboot your ST. The name of your accessory should appear under the "Desk" menu. If you select your accessory, it should run.

**CAUTIONS:**

(1) A desk accessory should NEVER "finish". That is, your event loop should not provide any way to exit the program and you should not call Exit_Gem.

(2) As you start writing desk accessories, you will likely find that they are harder to debug than normal programs. May we suggest that you first write and debug the bulk of your program as a standard GEM ".PRG" program before converting it to a desk accessory.

(3) Various pieces of GEM documentation say that msg[4] is supposed to contain the menu ID number for your accessory when you get an AC_Open or AC_Close message. (Especially for AC_Open this makes sense: the user clicked on that spot in the menu to activate your accessory.) However, under the version of GEM that we have at this writing, msg[4] is ALWAYS zero (0) for desk accessories. Because of this conflict, we have commented out some code in our sample accessory that checks the value of msg[4]. There were no ill effects, and the accessory runs fine. However, you should be aware of a possible conflict here in future versions of GEM.

**A LISTING OF ACCDEMO.PAS (SAMPLE ACCESSORY) IS ON YOUR Personal Pascal DISKETTE.**

# ERROR NUMBER DESCRIPTIONS

1 Error in Simple Type
2 Identifier expected
3 PROGRAM expected
4 ')' expected
5 ':' expected
6 Illegal symbol (possibly missing a ';' on the previous line)
7 Error in parameter list
8 OF expected
9 '(' expected
10 Error in Type
11 '[' expected
12 ']' expected
13 END expected
14 ';' expected (possibly on previous line)
15 Integer expected
16 '=' expected
17 BEGIN expected
18 Error in declaration part
19 Error in field list
20 '.' expected
21 '*' expected
50 Error in constant
51 ':=' expected
52 THEN expected
53 UNTIL expected
54 DO expected
55 TO or DOWNTO expected (in FOR statement)
56 IF expected (EXIT without IF)
57 EXIT expected
58 Error in expression
59 Error in variable
101 Identifier declared twice
102 Low bound exceeds high bound
103 Identifier not of appropriate class
104 Undeclared Identifier
105 Sign not allowed

## ERROR NUMBERS ( Continued )

106 Number expected
107 Incompatible subrange types
108 File not allowed here
109 Type must not be Real or Long_Integer
110 variant tag-type must be ordinal
111 Constant incompatible with tag-type
112 Index type can't be Real or Long_Integer
113 Index type must be ordinal
114 Base type must not be Real or Long_Integer
115 Base byte must be ordinal
116 Error in parameter type
117 Unsatisfied forward reference
118 Illegal forward reference of TYPE
119 Formal Parameters not allowed when completing a FORWARD subprogram
120 Function result must be ordinal or pointer
121 File value parameter not allowed
122 Illegal re-declaration of FORWARD result
123 Missing result type in Function header
124 Fractional digits format for Reals only
125 Error in type of parameter
126 Number of actual parameters does not match formal parameter declaration
127 Illegal parameter substitution
128 Result type does not agree with declaration
129 Type conflict of operands
130 Expression not of SET type
131 Only tests on equality allowed
132 Strict inclusion not allowed
133 File comparison not allowed
134 Illegal type of operand(s)
135 Operand type must be Boolean
136 SET base type must be ordinal
137 SET base types must be compatible
138 Variable not an ARRAY type
139 Index type not compatible with declaration
140 Variable not a RECORD type
141 Variable must be a FILE or pointer
142 Illegal parameter solution

## ERROR NUMBERS ( Continued )

143 FOR control variable must be ordinal
144 Illegal expression type
145 Type conflict
146 Assignment of FILEs not allowed
147 Label type incompatible with selector
148 Subrange bounds must be ordinal
149 Index type can't be Integer or Long_Integer
150 Assignment to standard function not allowed
151 Assignment to formal function not allowed
152 No such field in RECORD
153 Type error in Read
154 Actual parameter must be a variable
155 FOR control VAR can't be formal or non-local
156 Multi-defined CASE constant
157 Too many cases in CASE statement
158 No such variant in this RECORD
159 Variant tag field must be ordinal
160 Subprogram already defined
161 Subprogram declared FORWAR twice
162 Parameter size must be constant
163 Missing variant in declaration
164 standard subprograms may not be passed as subprogram parameters
165 Multi-defined label
166 Multi-declared label
167 Undeclared label
168 Undefined label
169 Too many members in SET's base type
170 Value parameter expected
171 Redeclaration of standard file
172 Undeclared external file
174 Pascal subprogram expected
175 Actual STRING parameter dimension less than formal variable parameter
176 Source STRING or substring larger than destination's dimension
177 Actual STRING parameter dimension greater than formal value parameter
178 Input of STRINGs with Readln only
179 STRING variable must be last parameter

## ERROR NUMBERS ( Continued )

| | |
|---|---|
| 194 | Compiler directive may appear only before PROGRAM declaration |
| 201 | Error in Real number - digit expected. |
| 202 | String constant must not exceed source line |
| 203 | Integer constant exceeds range |
| 250 | Too many scopes of nested identifiers |
| 251 | Too many nested subprograms |
| 252 | Too many forward references |
| 253 | Subprogram statement part too big (greater than 32 Kbytes) |
| 254 | Too many long constants in this subprogram |
| 256 | Too many external references |
| 257 | Too many externals |
| 258 | Too many local variables |
| 259 | Expression too complex |
| 300 | Division by zero |
| 301 | No case provided for this value |
| 302 | Index expression out of bounds |
| 303 | Value to be assigned out of bounds |
| 304 | Element expression out of range |
| 390 | Too much space alloc'd for global variables |
| 391 | Too much space alloc'd for local variables |
| 392 | Type too big |
| 393 | Too much space allocated for parameters |
| 394 | Language directive expected |
| 395 | Numbers outside range 0..127 not allowed |
| 396 | Parameter occupies more than two words |
| 398 | Implementation restriction |
| 399 | Implementation restriction |
| 400 | Illegal character in text |
| 401 | Unexpected end of input |
| 403 | Error in reading include file |
| 406 | Include file not legal |

# GEM KEYBOARD CODES

Whenever a GEM program gets a keyboard event, the code for the key causing the event is returned (see Get_Event). This Appendix lists all the possible keyboard codes.

Note that the codes are given in hexadecimal and that they are given as "high byte" and "low byte." This is in accordance with GEM documentation and usage. Further, note that the "low byte" value for most keys is the normal ASCII equivalent. For those keys or key combinations which have no ASCII representation, the lower byte will be zero. To distinguish between such keys--and to allow detection of such things as the numeric keypad keys--use the upper byte value.

The keycodes for the numeric keypad are not listed in Atari's documentation and were obtained through experimentation. It is possible that other keycodes exist which are as yet undocumented.

Note that if you call BIOS for a key (instead of waiting for a GEM keyboard event), then each byte of a keycode is returned in a separate word. The values for this "high word, low word" method remain the same.

| High | Low | Character |
|---|---|---|
| 03 | 00 | CNTL 2 |
| 1E | 01 | CNTL A |
| 30 | 02 | CNTL B |
| 2E | 03 | CNTL C |
| 20 | 04 | CNTL D |
| 12 | 05 | CNTL E |
| 21 | 06 | CNTL F |
| 22 | 07 | CNTL G |
| 23 | 08 | CNTL H |
| 17 | 09 | CNTL I |
| 24 | 0A | CNTL J |
| 25 | 0B | CNTL K |
| 26 | 0C | CNTL L |
| 32 | 0D | CNTL M |
| 31 | 0E | CNTL N |
| 18 | 0F | CNTL O |
| 19 | 10 | CNTL P |
| 10 | 11 | CNTL Q |
| 13 | 12 | CNTL R |
| 1F | 13 | CNTL S |
| 14 | 14 | CNTL T |
| 16 | 15 | CNTL U |
| 2F | 16 | CNTL V |
| 11 | 17 | CNTL W |
| 2D | 18 | CNTL X |
| 15 | 19 | CNTL Y |
| 2C | 1A | CNTL Z |
| 1A | 1B | CNTL [ |
| 2B | 1C | CNTL \ |
| 1B | 1D | CNTL ] |
| 07 | 1E | CNTL 6 |
| 0C | 1F | CNTL - |

| High | Low | Character |
|---|---|---|
| 39 | 20 | SPACE |
| 02 | 21 | ! |
| 28 | 22 | " |
| 04 | 23 | # |
| 05 | 24 | $ |
| 06 | 25 | % |
| 08 | 26 | & |
| 28 | 27 | ' |
| 0A | 28 | ( |
| 0B | 29 | ) |
| 09 | 2A | * |
| 0D | 2B | + |
| 33 | 2C | , |
| 0C | 2D | - |
| 34 | 2E | . |
| 35 | 2F | / |
| 0B | 30 | 0 |
| 02 | 31 | 1 |
| 03 | 32 | 2 |
| 04 | 33 | 3 |
| 05 | 34 | 4 |
| 06 | 35 | 5 |
| 07 | 36 | 6 |
| 08 | 37 | 7 |
| 09 | 38 | 8 |
| 0A | 39 | 9 |
| 27 | 3A | : |
| 27 | 3B | ; |
| 33 | 3C | < |
| 0D | 3D | = |
| 34 | 3E | > |
| 35 | 3F | ? |

| High | Low | Character |
|---|---|---|
| 03 | 40 | @ |
| 1E | 41 | A |
| 30 | 42 | B |
| 2E | 43 | C |
| 20 | 44 | D |
| 12 | 45 | E |
| 21 | 46 | F |
| 22 | 47 | G |
| 23 | 48 | H |
| 17 | 49 | I |
| 24 | 4A | J |
| 25 | 4B | K |
| 26 | 4C | L |
| 32 | 4D | M |
| 31 | 4E | N |
| 18 | 4F | O |
| 19 | 50 | P |
| 10 | 51 | Q |
| 13 | 52 | R |
| 1F | 53 | S |
| 14 | 54 | T |
| 16 | 55 | U |
| 2F | 56 | V |
| 11 | 57 | W |
| 2D | 58 | X |
| 15 | 59 | Y |
| 2C | 5A | Z |
| 1A | 5B | [ |
| 2B | 5C | \ |
| 1B | 5D | ] |
| 07 | 5E | ^ |
| 0C | 5F | _ |

| High | Low | Character |
|---|---|---|
| 29 | 60 | ` |
| 1E | 61 | a |
| 30 | 62 | b |
| 2E | 63 | c |
| 20 | 64 | d |
| 12 | 65 | e |
| 21 | 66 | f |
| 22 | 67 | g |
| 23 | 68 | h |
| 17 | 69 | i |
| 24 | 6A | j |
| 25 | 6B | k |
| 26 | 6C | l |
| 32 | 6D | m |
| 31 | 6E | n |
| 18 | 6F | o |
| 19 | 70 | p |
| 10 | 71 | q |
| 13 | 72 | r |
| 1F | 73 | s |
| 14 | 74 | t |
| 16 | 75 | u |
| 2F | 76 | v |
| 11 | 77 | w |
| 2D | 78 | x |
| 15 | 79 | y |
| 2C | 7A | z |
| 1A | 7B | { |
| 2B | 7C | \| |
| 1B | 7D | } |
| 29 | 7E | ~ |
| 0E | 7F | DEL |

| High | Low | Character |
|---|---|---|
| 81 | 00 | ALT 0 |
| 78 | 00 | ALT 1 |
| 79 | 00 | ALT 2 |
| 7A | 00 | ALT 3 |
| 7B | 00 | ALT 4 |
| 7C | 00 | ALT 5 |
| 7D | 00 | ALT 6 |
| 7E | 00 | ALT 7 |
| 7F | 00 | ALT 8 |
| 80 | 00 | ALT 9 |
| 1E | 00 | ALT A |
| 30 | 00 | ALT B |
| 2E | 00 | ALT C |
| 20 | 00 | ALT D |
| 12 | 00 | ALT E |
| 21 | 00 | ALT F |
| 22 | 00 | ALT G |
| 23 | 00 | ALT H |
| 17 | 00 | ALT I |
| 24 | 00 | ALT J |
| 25 | 00 | ALT K |
| 26 | 00 | ALT L |
| 32 | 00 | ALT M |
| 31 | 00 | ALT N |
| 18 | 00 | ALT O |
| 19 | 00 | ALT P |
| 10 | 00 | ALT Q |
| 13 | 00 | ALT R |
| 1F | 00 | ALT S |
| 14 | 00 | ALT T |
| 16 | 00 | ALT U |
| 2F | 00 | ALT V |
| 11 | 00 | ALT W |
| 2D | 00 | ALT X |
| 15 | 00 | ALT Y |
| 2C | 00 | ALT Z |

| High | Low | Character |
|---|---|---|
| 3B | 00 | F1 |
| 3C | 00 | F2 |
| 3D | 00 | F3 |
| 3E | 00 | F4 |
| 3F | 00 | F5 |
| 40 | 00 | F6 |
| 41 | 00 | F7 |
| 42 | 00 | F8 |
| 43 | 00 | F9 |
| 44 | 00 | F10 |
| 54 | 00 | SHIFT F1 |
| 55 | 00 | SHIFT F2 |
| 56 | 00 | SHIFT F3 |
| 57 | 00 | SHIFT F4 |
| 58 | 00 | SHIFT F5 |
| 59 | 00 | SHIFT F6 |
| 5A | 00 | SHIFT F7 |
| 5B | 00 | SHIFT F8 |
| 5C | 00 | SHIFT F9 |
| 5D | 00 | SHIFT F10 |
| 73 | 00 | CNTL LEFT ARROW |
| 4D | 00 | RIGHT ARROW |
| 4D | 36 | SHIFT RIGHT ARROW |
| 74 | 00 | CNTL RIGHT ARROW |
| 50 | 00 | DOWN ARROW |
| 50 | 32 | SHIFT DOWN ARROW |
| 48 | 00 | UP ARROW |
| 48 | 38 | SHIFT UP ARROW |
| 77 | 00 | CNTL HOME |
| 47 | 00 | HOME |
| 47 | 37 | SHIFT HOME |
| 52 | 00 | INSERT |
| 52 | 30 | SHIFT INSERT |
| 53 | 00 | DELETE |
| 53 | 2E | SHIFT DELETE |
| 01 | 1B | ESC |

| High | Low | Character | High | Low | Character |
|---|---|---|---|---|---|
| 0E | 08 | BACKSPACE | 82 | 00 | ALT - |
| 83 | 00 | ALT = | 1C | 0D | RETURN |
| 1C | 0A | CNTL RETURN | 0F | 09 | TAB |
| 4B | 00 | LEFT ARROW | 4B | 34 | SHIFT LEFT ARROW |
| 62 | 00 | HELP | 61 | 00 | UNDO |
| 39 | 00 | CNTL SPACE | | | |

(the following codes are from the numeric keypad)

| | | | | | |
|---|---|---|---|---|---|
| 70 | 30 | 0 | 70 | 10 | CNTL 0 |
| 6D | 31 | 1 | 6D | 11 | CNTL 1 |
| 6E | 32 | 2 | 6E | 00 | CNTL 2 |
| 6F | 33 | 3 | 6F | 13 | CNTL 3 |
| 6A | 34 | 4 | 6A | 17 | CNTL 4 |
| 6B | 35 | 5 | 6B | 15 | CNTL 5 |
| 6C | 36 | 6 | 6C | 1E | CNTL 6 |
| 67 | 37 | 7 | 67 | 17 | CNTL 7 |
| 68 | 38 | 8 | 68 | 18 | CNTL 8 |
| 69 | 39 | 9 | 69 | 19 | CNTL 9 |
| 4A | 2D | - | 4A | 1F | CNTL - |
| 4E | 2B | + | 4E | 0B | CNTL + |
| 63 | 28 | ( | 63 | 08 | CNTL ( |
| 64 | 29 | ) | 64 | 09 | CNTL ) |
| 65 | 2F | / | 65 | 0F | CNTL / |
| 66 | 2A | * | 66 | 0A | CNTL * |
| 71 | 2E | . | 71 | 0E | CNTL . |
| 72 | 0D | ENTER | 72 | 0A | CNTL ENTER |

# INDEX

**NOTE:**

Names in italics are library functions or procedures.
+ means "and next page"
++ means "and successive pages"

# NOTES AND CAUTIONS

This index to the notes and cautions is not exhaustive; we have included only those we thought to be of greatest importance to the programmer.

**NOTES:**

## CAUTIONS:

# TACKLE BOX ST

Tackle Box ST consists of a 3-ring binder holding 900 pages of documentation and three diskettes, all packed full of of helpful hints, ready-to-use subroutines, and complete access to all the best features of the ST, including full GEM support.

Truthfully, this package is so good and so complete that it would have to sell for $150 if it were sold through normal channels. Because the producer, SRM Enterprises, is a new and small company, they are offering it for direct sale at the very reasonable price of $69.95.

Does that sound like a lot? Then compare it to the cost of 2 or 3 Abacus books (at $20 each), a handful of example disks ($15 each?), and a lot of your own time, trying to translate the documentation into Pascal terms. If you want to get the most out your ST, then Tackle Box ST is for you.

OSS has arranged with SRM Enterprises to take orders for Tackle Box ST. Plus, if you order it through OSS, we will throw in a demonstration disk, full of utility programs, perhaps a game or two, maybe a few surprises. All, of course, in full Pascal source code form.

# TO ORDER:

| | |
|---|---|
| **Tackle Box ST** | **$69.95** |
| **IL Sales Tax** | **$4.37** |
| *(for IL residents only)* | |
| **Shipping** | |
| ***Continental U.S.*** | **$6.00** |
| ***Canada, Alaska, Hawaii*** | **$10.00** |

***Send To:***

**ICD/OSS, Inc.**
**1220 Rock Street**
**Rockford, IL 61101**

We also accept Visa , MasterCard and C.O.D. orders.
Phone orders are welcome Monday-Friday, 8A.M. to 5P.M. CST.
**(815) 968-2228**

BLANK
PAGE

## Personal Pascal

Turn your Atari ST computer into a "powerhouse" with **Personal Pascal**, the hottest selling language for Atari ST computers. That's right! **Personal Pascal** provides you with all the tools necessary to produce Commercial programs. Or, enjoy programming in **Personal Pascal** as a hobby.

**Personal Pascal** goes beyond standard Pascal by providing you with commands that take full advantage of your Atari ST and its built-in GEM operating system.

A special **Personal Pascal** Library call provides you with Turbo-compatible screen procedures. As packaged, **Personal Pascal** comes complete with Editor, Compiler, Linker and Libraries. Make **Personal Pascal** your choice for programming and discover the hidden powers of your Atari ST.

No additional license fees for Commercially produced programs written in **Personal Pascal**.