ORCA/Pascal 2.0™

An Object Pascal Compiler and Development System for the Apple IIGS

Mike Westerfield

Byte Works®, Inc.
4700 Irving Blvd. NW, Suite 207
Albuquerque, NM 87114
(505) 898-8183

Credits

Pascal Compiler
Mike Westerfield

Development Environment
Mike Westerfield
Phil Montoya

Testing
Mike Westerfield

Documentation
Mike Westerfield

Table of Contents

Chapter 1 – Introducing ORCA/Pascal

ORCA/Pascal

Welcome to ORCA/Pascal! ORCA/Pascal is a complete, stand-alone program containing all of the software you need to write professional quality programs on the Apple IIGS. The package includes a fast, easy to use Pascal compiler, a linker that lets you create and use libraries, or even mix Pascal programs with subroutines written in other languages, and two complete development environments. This manual is based on the most popular of the two development environments, which we refer to as the desktop development environment. The desktop development environment gives you fast graphics and mouse based editing. The editor supports files up to the size of available memory; split screen; search and replace; cut, copy and paste; the ability to edit several files at one time; and several specialized editing features. The desktop development environment also features a built-in debugger. This source level debugger lets you debug Pascal programs, showing you what line is executing and the values of the variables. It supports many advanced debugging features like step-and-trace, break points, and profiling.

The second development environment is a UNIX-style text based development environment. This is an updated version of the same environment sold by Apple Computer as Apple Programmer's Workshop (APW). Many programmers who program on a daily basis prefer text environments for their speed and power. In later chapters, you will learn how to set up and use the text based environment. At least while you are getting started, we recommend using the desktop development environment unless you have a compelling reason to use the text environment. You might want to consider the text environment if you are working on a computer without much memory, or if you are used to text environments and prefer them over desktop programs.

In later chapters, as we explore the capabilities of the desktop environment, you will also find that the power of the text based shell is not lost to those who prefer the desktop environment. The central part of the text based environment is a powerful, programmable shell. The shell is a program that gives you control over the files on your disks, the process of compiling programs, and where program output goes and input comes from. You may have used simple shells before, like BASIC.SYSTEM, used with AppleSoft. The ORCA shell shares many features with these simpler shells, but is much more powerful.

After purchasing a new program, you would probably like to sit right down at your computer and try it out. We encourage you to do just that, and in fact, this manual is designed to help you. Before getting started, though, we would like to take some time to suggest how you should approach learning to use ORCA/Pascal, since the best approach is different for different people.

What You Need

To use ORCA/Pascal, you will need an Apple IIGS with at least 1.125M of memory for ROM 3 machines, or 1.25M of memory with ROM 1 machines. You will also need at least two disk drives, and at least one of those must be a 3.5" disk drive. To use all of the features and utilities included with ORCA/Pascal, you will need at least 1.75M of memory and a hard disk; this is the minimum system for developing desktop programs.

You do not need to do any initialization to use ORCA/Pascal. Simply insert the 3.5" disk labeled "Apple IIGS System Disk" in your 3.5" disk and boot your computer. After a few moments, the Finder will appear. Eject the boot disk and insert the disk labeled "Program Disk." Execute ORCA.Sys16, which shows up with an orca whale icon. If you are unfamiliar with the basic operation of your computer, refer to the manuals supplied with the computer itself.

If you have a hard disk, or if you have two 3.5" floppies and would like to use ORCA/Pascal with its programmable shell or in combination with another language, you will want to reconfigure your system. The Extras Disk has a copy of Apple's installer, along with several scripts that will help you install ORCA/Pascal in a variety of different configurations; these are explained in detail in Appendix C. While these installation instructions make it easy to install ORCA/Pascal, the first few chapters of this manual assume you are running the program as we ship it. There may be a few points that will be confusing to you until you have learned a little more about ORCA/Pascal; if you find that you are confused, you might try using the floppy-disk based system until you learn how to use ORCA/Pascal.

About the Manual

This manual is your guide to ORCA/Pascal. To make it easy for you to learn about the system, this manual has been divided into three major sections. The first part is called the "User's Guide." It is a tutorial introduction to the development environment, showing you how to create Pascal programs under ORCA. The second part is called the "Environment Reference Manual." It is a working reference to provide you with in-depth information about the development environment you will use to create and test Pascal programs. Part three is the "Language Reference Manual." It contains information about the ORCA/Pascal programming language. This organization also makes it easy for you to skip sections that cover material that you already know. For example, the ORCA languages are unique on the Apple IIGS in that a single development environment can be used with many different languages. If you have already used the development environment with another ORCA language, you can skip the sections that cover the environment, and concentrate on the Pascal programming language.

While this manual will teach you how to use ORCA/Pascal to write and test programs, it does not teach you the basics of the language itself. Basic concepts about programming in Pascal are necessary to create useful, efficient programs. If you are new to Pascal, you can start with our Learn to Program in Pascal course, which is written specifically for ORCA/Pascal on the Apple IIGS. You’ll find some details about this course, and several other books that may be of interest, at the end of this chapter.

If you are new to ORCA, start at the beginning and carefully read the first three chapters of the "User's Manual," along with any portions of Chapter 4 that interest you. These sections were written with you in mind. Work all the examples, and be sure that you understand the material in each chapter before leaving it. ORCA is a big system, and like any sophisticated tool, it takes time to master. On the other hand, you don't need to know everything there is to know about ORCA to create sophisticated programs, and the desktop environment makes it easy to write and test the most common kinds of Pascal programs. The first four chapters give you enough information to create, test and debug Pascal programs using ORCA/Pascal. After working through these chapters, you can skim through the rest of the manual to pick up more advanced features.

From time to time, we make improvements to ORCA/Pascal. You should return your registration card so we can notify you when the software is improved. We also notify our customers when we release new products, often offering substantial discounts to those who already have one of our programs.

Visual Cues

In order to tell the difference between information that this manual provides and characters that you type or characters that appear on your computer screen, special type faces are used. When you are to enter characters, the type face **`looks like this.`** When you are supposed to notice characters displayed on the computer screen `they look like this.` Named keys, such as the return key, are shown in outline, like this.

ISO ORCA/Pascal is a superset of the ISO Pascal Standard. When the manual discusses extensions to the standard, you'll see a bullet like this one. ∆

Other Books and Reference Materials

If you are new to Pascal, you will need to supplement this manual with a good beginner's book on the Pascal programming language. A companion course is available from the Byte Works that teaches you the Pascal language and some basic techniques for programming. The book is called Learn to Program in Pascal, and it has one distinct advantage over any other Pascal programming book: it is written specifically for ORCA/Pascal running on an Apple IIGS. Another good way to find a book that suits you, especially if you already know a little Pascal, is to visit a well-stocked bookstore and look through their selection. There are literally hundreds of books that cover various aspects of programming in Pascal; which book you choose depends on your background. A few of our favorites are listed below.

If you will be using the Apple IIGS Toolbox to create your own desktop programs, you should have a copy of the Apple IIGS Toolbox Reference, volumes 1 through 3, and Programmer's Reference for System 6.0. These books do not teach you about the toolbox, but they are essential references. For an introduction to the toolbox, we suggest Toolbox Programming in Pascal, which is a complete introduction to the world of toolbox programming.

Learn to Program in Pascal
Mike Westerfield
Byte Works, Inc., Albuquerque, New Mexico
This introductory Pascal programming course is written specifically for ORCA/Pascal running on an Apple IIGS. It contains hundreds of complete programs as examples, as well as problems with solutions.

Toolbox Programming in Pascal
Mike Westerfield
Byte Works, Inc., Albuquerque, New Mexico
This is the only self-paced course available for programming the Apple IIGS toolbox. Unlike the toolbox reference manuals, this is a course that teaches you how to write programs, not a catalog of the various toolbox calls available on the Apple IIGS. It includes four disks filled with toolbox source code, as well as an abridged toolbox reference manual, so you won't have to buy all of the toolbox reference manuals right away.

Oh! Pascal!
Doug Cooper and Michael Clancy
W.W. Norton & Company, New York, NY
Our favorite Pascal textbook.

Standard Pascal User Reference Manual
Doug Cooper
W.W. Norton & Company, New York, NY
Our favorite Pascal reference manual.

Technical Introduction to the Apple IIGS
Apple Computer
Addison-Wesley Publishing Company, Inc. Reading, MA
A good basic reference source for the Apple IIGS.

Apple IIGS Hardware Reference and Apple IIGS Firmware Reference
Apple Computer
Addison-Wesley Publishing Company, Inc. Reading, MA
These manuals provide information on how the Apple IIGS works.

Programmer's Introduction to the Apple IIGS
Apple Computer
Addison-Wesley Publishing Company, Inc. Reading, MA
Provides programming concepts about the Apple IIGS.

Apple IIGS Toolbox Reference: Volume I, Apple IIGS Toolbox Reference: Volume II and Apple IIGS Toolbox Reference: Volume III
Apple Computer
Addison-Wesley Publishing Company, Inc. Reading, MA
These volumes provide essential information on how the tools work – the parameters you need to set up and pass, the calls that are available, etc. You must have these books to use the Apple IIGS toolbox effectively.

Programmer's Reference for System 6.0
Mike Westerfield
Byte Works, Inc., Albuquerque, New Mexico
The first three volumes of the toolbox reference manual cover the Apple IIGS toolbox up through System 5. This book covers the new features added to the toolbox and GS/OS in System 6.

GS/OS Reference
Apple Computer
Addison-Wesley Publishing Company, Inc. Reading, MA
This manual provides information on the underlying disk operating system. It is rarely needed for Pascal programming, since Pascal has built-in subroutines for dealing with disk files.

ORCA/M A Macro Assembler for the Apple IIGS
Mike Westerfield and Phil Montoya
Byte Works, Inc., Albuquerque, NM
ORCA/M is a macro assembler that can be used with ORCA/Pascal. Without changing programming environments, you can create a program in Pascal, assembly language, or a combination of the two. Chapter 5 will give you more information on how easy it is to mix the two languages.
ORCA/C
Mike Westerfield
Byte Works, Inc., Albuquerque, NM
ORCA/C is a C compiler which can be installed in the same environment as ORCA/Pascal. With the C compiler installed, you can write C or Pascal programs without switching environments. You can also use library routines written in C from your Pascal programs.

Chapter 2 – Exploring the System

Backing Up ORCA/Pascal

This chapter is a hands-on introduction to ORCA/Pascal. You should read it while seated at your computer, and try the things suggested as we talk about them. By the end of the chapter, you will have a good general feel for what Pascal programming is like using ORCA/Pascal. The next two chapters introduce slightly more advanced topics, including control of the compiler, and how to write programs for the various environments supported on the Apple IIGS.

As with any program, the first step you should take is to make a backup copy of the original disks. To do this, you will need five blank disks and a copy program – Apple's Finder, from the System Disk, will do the job, or you can use any other copy program if you have a personal favorite. If you are unfamiliar with copying disks, refer to the documentation that came with your computer. As always, copies are for your personal use only. Using the copies for any purpose besides backing up your program is a violation of federal copyright laws. If you will be using ORCA/Pascal in a classroom or work situation where more than one copy is needed, please contact the publisher for details on our licensing policies.

The Bull's Eye Program

If you have not yet made a backup, please do so at this time. The steps we will take in this section will change the disk.

The first thing we will do is run a simple sample program that draws a bull's eye on the screen. We will use this program to get an overview of the system, and gradually build on this foundation by supplying more and more details about what is happening. Begin by making sure that all of the disks you are using are not write protected, since ORCA/Pascal does write some information to disk as it prepares your programs. The first step is to start ORCA/Pascal. To do this, insert a copy of the disk labeled "Apple IIGS System Disk" into your computer and boot the disk, which will boot into Apple's Finder. Eject this disk and insert the "Program Disk," and launch the program called ORCA.Sys16 – the one that shows up with the orca whale icon. After a few moments, you will see a standard desktop – you are ready to start using ORCA/Pascal.

Go ahead and select Open from the File menu. In the list of files you will see a folder called Samples. Open this folder by clicking twice in rapid succession on the name, or by clicking once on the name to select the folder, and then clicking on the Open button. You will see another, shorter list of files. One of these is called BullsEye.pas. This file is the source code for the Pascal program we will run. Click twice on the file name, and the program will appear in a window on the desktop.

The bull's eye program will draw several circles, one inside the other. ORCA/Pascal let's you see the output from your program while you look at the source code. Naturally, to do this, you need someplace to put the output. In the case of graphics output, the drawing appears in a special window called the Graphics Window. To see this window, you need to do two things. First, shrink the bull's eye program's window by holding the mouse down in the grow box (the box at the bottom right of the window) and dragging the grow box to the left. You want to cut the width of the window to about half of the screen, so the right side of the window is just before the start of the word Run in the menu bar. Now pull down the Windows menu and select the Graphics Window command. The graphics window will show up in the lower right portion of the screen.

Positioning the windows is the hard part! To run the program, pull down the Run menu and select the Compile to Memory command. A third window, called Shell, will show up in the top right portion on the screen. The system uses this window to write text error messages and keep you informed about progress as the program is compiled and linked. The first compile of the day takes a little time, so be patient. The desktop development environment is a multi-lingual programming environment. Because the program doesn't know in advance what language you will be using, it waits until you compile a program to load the compiler and linker. If you have 1.75M of memory, and haven't set aside a large RAM disk, these programs remain in memory, so subsequent compiles are much faster. In addition, once a program has been compiled, if you try to compile it again without changing the source file, the program is simply executed. To see this, try the Compile To Memory command again. The executable program is loaded from disk and re-executed.

You might wonder why the executable program is saved to disk when you use the Compile To Memory command. Compile To Memory refers to the intermediate files, called object modules, that are passed from the compiler to the linker when your program is prepared for execution. For some advanced applications, you will want to save these to disk, but for simple programs like the bull's eye program, the Compile To Memory command gives you faster compiles by not writing the object modules to disk. For both Compile to Memory and Compile to Disk, though, the executable program is still saved on the disk.

Before moving on, let's try one more command. Pull down the Debug menu and select Trace. Watch the left margin of your source window as the program runs – you will see an arrow moving from line to line in the source code. The ability to trace through a program is the foundation of the debugger supplied with ORCA/Pascal. In later sections, we will explore this capability in detail.

Finding Out About the Desktop

As you can see, it's pretty easy to load, compile and execute programs using ORCA/Pascal. One of the main advantages of the desktop programming environment is ease of use. The rest of this chapter explains how to use the desktop development environment to develop programs, but it assumes that you already know how to use menus, how to manipulate windows on the desktop, and how to edit text using a mouse. If you had any trouble understanding how to use the mouse to manipulate the menu commands and window in the last section, or if you are unfamiliar with mouse-based editors, now would be a good time to refer to Chapter 3 of the Apple IIGS Owner's Guide, which came with your computer. The owner's guide has a brief tutorial introduction to using desktop programs. Complete details on our desktop can be found in Chapter 7 of this manual, but that chapter is arranged for reference – if you are completely new to desktop programs, a gentler introduction, like the one in the user's guide, is probably better. The major features of our desktop development environment that are specific to programming, and are therefore not covered in Apple's introductory manual, will be covered in the remainder of this chapter.

How Graphics and Text are Handled

One of the unique features of the desktop development environment is its ability to show you the program and its output at the same time. You have already seen an example of this. The bull's eye program produces graphics and text output (it writes the string "Bull's eye!" after the bull's eye is drawn). Most books on Pascal teach you the language using text input and output. As with the bull's eye program, the text will show up in a special window called the Shell window. This window is created automatically when you compile a program, and stays around until you close it. You can resize it – even hide it behind the program window if it bothers you.

The shell window is used for several other purposes besides giving your program a place to write text output. If your program needs input from the keyboard, you will see the input echoed in the shell window. The compiler and linker also write error messages to the shell window. These error messages will also be shown in a dialog, so you won't miss the error even if you hide the shell window. Writing the errors to the shell window, though, gives you a more permanent record of the errors. Later, in Chapter 6, we will explore still more uses of the shell window.

The graphics window lets you write programs that draw pictures without doing all of the initialization required to write stand-alone programs. For example, if you are writing a game that uses pull-down menus and windows, your program will open windows for itself. For simple graphics tasks like drawing bull's eyes or plotting a function, though, the graphics window lets you concentrate on the algorithms and on the graphics language, without all of the fuss of learning how to create menus and windows for yourself.

You don't need to open the graphics window unless your program uses it. If you want to use the graphics window, just be sure to open it before running your program. (If you forget, nothing tragic happens – you just won't be able to see the graphics output from your program.) If you need more space, you can drag the window around and size it.

One feature of the graphics window is worth pointing out. When your program draws to the graphics window, it does so using QuickDraw II. The development environment does not know what commands you are using, so it cannot repaint the window. What this means is that if you move a window on top of a drawing in the graphics window, and then move it back off, the only way to refresh the picture is to run the program again. You might try this right now to see what we mean. Drag the shell window down so it covers about half of the graphics window, then move it back to its original location. The part of the graphics window that was covered will be erased.

The Languages Menu

The Languages menu shows all of the languages installed on your system. It changes when you install or delete a programming language. You can use this menu to find out what language is associated with a particular file, or to change the language.

Under ORCA, all source and data files are associated with a language. The system uses the underlying language stamp to call the appropriate compiler or assembler when you issue a compile command for a source file. For example, if you select the BullsEye.pas source file (a window is selected by clicking anywhere on the window) and pull down the Languages menu, you will see PASCAL checked. If you select the shell window, the language SHELL will be checked. When you create a new program, the system tries to select the proper language automatically by assigning the language of the last file edited. You should always check the language menu, though. If you write a Pascal program, and the system thinks it is an assembly language source file, the assembler will give you enough errors that you will know something is wrong. If you don't have the assembler on the disk, a dialog will appear with the message "A compiler is not available for this language." In either case, simply pull down the Languages menu and select the appropriate language, then try compiling again.

What's a Debugger?

A debugger helps you find errors in your program. You can use a debugger to execute all or part of your program one line at a time, watching the values of selected variables change as the program runs. If you know that some subroutines are working, while there are problems with other subroutines, you can execute the working routines at full speed and then trace slowly through the problem areas. You can also set break points in your program and then have the debugger execute your program until it reaches the break.

While the desktop development system supports many languages besides Pascal, not all languages that work with the development system support the source-level debugger. If you are using another language with ORCA/Pascal, and are not sure whether or not it supports the debugger, try it. If the language doesn't support the debugger, your program will simply execute at full speed.

There is one very important point to keep in mind about the debugger. When you compile a program with debug on, the compiler inserts special code into your program to help the debugger decide which line it is on, where symbols are located, and so forth. If you run a program with debug code in it from the Finder or the text-based shell, the program will crash. For that reason, it is very important that you turn the debug option off after a program is finished. To turn debugging off, pull down the Run menu and select the Compile command. The dialog that appears has an option with the caption "Generate debug code." If there is an X in the box to the left of this option, debug code is turned on; if there is no X, it is turned off. Clicking in the box turns the option on and off. Once you set this option the way you want it, click on the Set Options button.

One other point about debug code deserves to be mentioned. The debug code takes time and space. When you turn debugging off, your program will get smaller and faster. In programs that do lots of graphics or floating point calculations, like the bull's eye program, the difference is relatively small, but in programs that spend their time looping and doing logical operations, the difference in execution speed can be considerable.

In the remainder of this chapter, we will look at how you can use the source-level debugger to find problems in your Pascal programs. The examples we will use here are fairly short, simple programs. You can debug large programs, including desktop applications. The basic ideas are similar, but there are a few restrictions to keep in mind. Debugging desktop programs is covered in a special section in Chapter 4.

Using the Source-Level Debugger

Let's use the bull's eye program again to become familiar with the source-level debugger. If you do not have the program open on the desktop, please pull down the File menu and use the Open command to load it from the Samples folder. Now shrink the bull's eye window to about half its current width, as before. If you do not have a graphics window open, pull down the Windows menu and use the Graphics Window command to open a graphics window.

Debugging a Simple Graphics Program

Pull down the Run menu and select the Compile command. The desktop brings up a dialog box. For now, just ignore all of the items in the Compile window except the box in front of the "Generate debug code" option. This box should be marked with an X, telling the compiler to produce the special code needed during debugging. After checking the "Generate debug code" box, click on the Set Options button at the bottom of the Compile window.

Now pull down the Debug menu.

The Step Command

Select the Step command from the Debug menu and watch the source file window. When the program starts to run, you will see an arrow pointing to the first line in the source file. Select Step again – the arrow now moves down to the second line in the program. You can continue to select Step from the Debug menu, or you can use the keyboard equivalent. Holding down the key and typing **[** will also step one time. Remembering the keystroke will be hard at first, but you can always pull down the menu to check to see what key is used: the key is shown to the right of the menu command name. Either way, each time you step, the arrow moves to the next line in the program, and the bull's eye is slowly painted on the graphics window, one circle at a time.

The Stop Command

Any time your program is executing, you can use the Stop command to stop the program. This also works when the debugger is paused, waiting for you to select the next debugging command.

The Trace Command

At any time, you can trace your program's execution by selecting Trace from the Debug menu. Once it starts tracing, the program will run until it finishes, or until you issue another debugging command. Select Trace from the Debug menu, and notice the arrow in the source file window – it moves through the lines of code as each line is executed. Any of the windows which might be open as a result of debugging (the source file, shell, variables, stack, and memory windows) will be continually updated while Trace is running.

To pause for a moment in the middle of a trace, move the cursor to the menu bar and press on the mouse button. You do not have to be on a menu; in fact, it is better if you aren't. As long as you hold down on the mouse button, the program will pause. When you let up, execution continues. While you have the mouse button down, if you decide to switch to step mode or stop the program, move to the Debug menu and select the appropriate command, or use the appropriate keyboard equivalent.

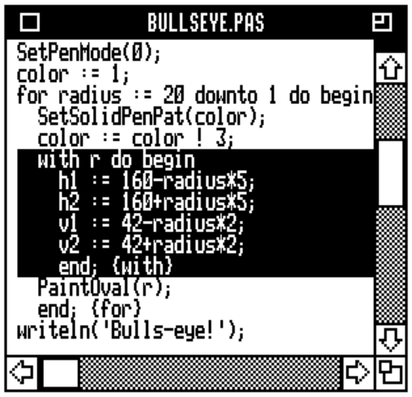
The Go Command

Experiment with the Go command in the Debug menu. It is similar to Trace, but executes an entire program at full speed. Unlike Trace, however, the debugging windows are not updated. Go is especially useful for quickly seeing the results of changing your program while you are fixing bugs. It is also useful when you are using break points and want to execute up to the location of the first break point.

Once a program is executing, it can be stopped by using one of the debug commands in the first part of the Debug menu. A break point or run-time error will also stop the program. You can pause debugging at any time by moving the cursor to the right-hand area of the menu bar and pressing on the mouse button. Debugging continues as soon as you release the mouse button.

The Set Auto-Go Command

Now let's look at Auto-Go. You can set lines for Auto-Go so that they will be executed at full speed, even if you are stepping or tracing. Use the mouse to select the four lines assigning values to the rectangle, as shown in the figure. Next, pull down the Debug menu and choose Set/Clear Auto-Go. A large green dot will appear to the left of each of the selected lines. Now use the Step command to step through the for loop. Notice that when the arrow stepped into the block of statements you selected, it jumped to the end of the block marked for auto-go. As you can see, Auto-Go can be very useful when you are stepping through your program, but don't want to see portions you have already debugged.



```
SetPenMode(0);
color := 1;
for radius := 20 downto 1 do begin
  SetSolidPenPat(color);
  color := color ! 3;
  with r do begin
    h1 := 160-radius*5;
    h2 := 160+radius*5;
    v1 := 42-radius*2;
    v2 := 42+radius*2;
    end; {with}
  PaintOval(r);
  end; {for}
writeln('Bulls-eye!');
```

Break Points

Next let's look at how to set break points. First use the Stop command to stop the program (if it hasn't already completed), and then select the program line containing the call to PaintOval. Now choose Set/Clear Break Point from the Debug menu. A purple X will appear to the left of the PaintOval line, indicating it is a break point. Now select Trace from the Debug menu. Execution stops at the PaintOval line.

A break point will always cause the program to stop – even if it was executing at full speed. Break points are especially useful for debugging large programs. You can set a break point on the first line of the area you want to examine, then execute the rest of the program at full speed. Execution will be suspended when you reach the break point.

Another use of break points is when you suspect that a certain portion of your program is not being executed at all. By setting a break point, you can check where your program quits executing, and then determine if this is in the location that you thought was not being reached.

Debugging a Program With More Than One Subroutine

There are several features of the debugger that are only useful in programs that have more than one subroutine. The bull's eye program we have been using so far doesn't have any subroutines, so we will need to switch to a program that does. If you haven't already done so, stop the bull's eye program. After you get the main menu back, close the graphics window and the bull's eye program's source window, and then open the file Sort.pas. Like the bull's eye program, the sort program is in the Samples folder. The sort program compares two simple sort procedures by sorting the same array of integers using each routine.

The Profile Command

One of the advanced features of the debugger that can help you improve a program is the profiler. The profiler collects statistics about your program to help you find bugs and "hot spots." A hot spot is a place in your program that takes a long time to execute compared with the rest of the program. You may have heard of a famous rule of thumb in programming which states that a program spends 90% of its time in 10% of its code. The 10% of the code is the hot spot, and knowing where it is can help you speed up your program.

As you can see, the sort program you just opened has two subroutines, named ShellSort and BubbleSort. Shrink the window to about half its width. Pull down the Debug menu and select the Profile command. This turns the profiler on. Next, use the Compile to Memory command to compile and execute the program, just as you did with the bull's eye program. After the program compiles and executes, you will see the profiler's statistics printed in the shell window. The profiler returns the following information:

1. The number of times each subroutine and main program was called.
2. The number of heartbeats while in each subroutine and the main program.

3. The percent of heartbeats for each subroutine and main program compared to the total number of heartbeats.

This information is in columns, and won't all be visible unless you expand the size of the shell window. If you don't see three columns of numbers after the names of the subroutines, make the shell window larger.

The number of times a subroutine is called is more useful than it seems at first. For example, let's say you are testing a program that reads characters from a file and processes them – a spelling checker, perhaps. If you know that the test file has 3278 characters, but the subroutine you call to read a single character is called 3289 times, you know right away that there is a problem. In addition, if you are really calling a subroutine 3278 times, and the subroutine is a short one that is only called from a few places, you might want to consider placing the few lines of code "in-line," replacing the subroutine calls. Your program will get larger, and perhaps a little harder to read, but the improvement in execution speed could make these inconveniences worthwhile.

The sort program only calls each sort one time, so the first column of information isn't very useful in this example. We also see, however, that the sort program spent about 32% of its time in the BubbleSort subroutine, about 42% of its time in the ShellSort routine, and about 26% of its time in the main program. At least for this type of data, then, the bubble sort is the better choice. You should be aware that the statistics generated by the profiler are based on a random sampling. It can be quite accurate for some types of programs, and very unreliable for others. To get the best results, run a program several times, and try to use input data that will cause it to execute for several seconds to a few minutes. The larger the sample, the better the results will be.

The Step Through Command

Two commands, Step Through and Goto Next Return, are designed to make debugging subroutines easier. The Step Through command is used to execute subroutines at full speed. For instance, many times when you are writing a new program, you may have problems with one or more of the subroutines, but you know that other subroutines are working fine. You would like to be able to pass quickly through the working routines, and then slow down and step through the problem areas of the code. This is the reason for the Step Through command.

To see how the Step Through command works, let's debug the Sort.pas program. If you pull down the Debug menu, you will see that the Step Through, Go to Next Return, and Stop items are all dimmed, meaning that they cannot be selected at this time. This is because there is nothing to step through or stop, and no return to go to.

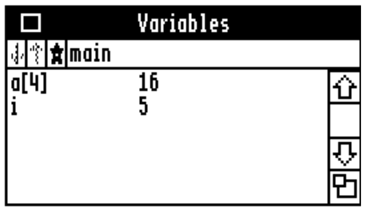
Pull down the Debug menu and select the Step command. Sort.pas is compiled and linked, and then our step arrow appears next to the for loop, which is the first statement in the main program. To get beyond the for loop, select Step eleven times. The step arrow is now next to the line containing the call to the ShellSort subroutine. Now pull down the Debug menu and select Step Through. There is a momentary pause, and then the arrow advances to the next line, another for loop. The Step Through command has just executed the ShellSort subroutine at full speed. If we now single-step through the for loop, we will see the sorted array values printed in the shell window.

The Goto Next Return Command

The Goto Next Return command is useful when you are only debugging a portion of a subroutine. To see how this command works, single-step through the statements in the main program until you reach the line containing the call to the BubbleSort routine. Single-step once more to reach the beginning of the BubbleSort subroutine. Now select Go to Next Return from the Debug menu. The BubbleSort routine is executed, and then the step arrow appears to the left of the line following that which called the BubbleSort function. To verify execution of the subroutine, we could use Step, Step Through, Go, or Trace to see the sorted array displayed in the shell window.

Viewing Program Variables

Watching a program execute, and seeing exactly when output is produced, can be very useful. The debugger has another ability, though, which is even more important: you can watch the values of the internal variables.



```
a[4]        16
i           5
```

To see how this works, pull down the Windows menu and select the Variables command. The desktop brings up a Variables window in the center of the screen, like the one pictured to the right. (The window you will see won't have any variable names in it.)

The rectangle beneath the title bar of the Variables window contains three boxes, and an area to the right of the boxes where the name of the currently executing subroutine is displayed. Drag the Variables window out of the way of the other windows on the desktop, and then select the source file window.

We can't enter any variables into the Variables window unless we are executing a program. This makes intuitive sense – memory for variables isn't allocated until run-time. The first two boxes control which subroutine we are looking at, while the third is a command button that displays all of the simple variables. Likewise, these boxes are dimmed until they can be used.

To see how to set up the variables we would like to view, start stepping through the program by using the Step command. You should be at the first line in the main program. Click anywhere in the Variables window below the title bar and function-name bar, and to the left of the scroll control. (This area is called the content region of the window.) A line-edit box will appear, with a flashing insertion point. Let's enter one of the main program's variables, a[4], and then press return. After the carriage return, we see the current value of a[4] displayed, which is zero.

You can enter new variable names by clicking in the content region of the Variables window, and then typing in the name. You can change an existing variable in the window by clicking on its name, and then using the line editor to make the necessary modifications.

Typing the name of a variable works great when we are trying to look at very specific things, like a particular element of an array, or if we just want to look at a few variables. It’s a little tedious to type the names of each and every variable, though. If you click on the third box – the one with a star – all of the simple variables will be displayed in the window. For arrays, records or pointers, though, you still have to type the specific value you want to see.

Continue stepping through the program, and watch what happens when the program enters the ShellSort procedure. The name ShellSort appears in the information bar of the Variables window, the up-arrow can be selected, and the variable a[4] vanishes. If you click on the up arrow to the left of the ShellSort procedure name, you will see the variables display for the main program, and the down arrow in the Variables window can be selected. The variable a[4], which is defined at the main program level, also reappears. If you click on the down arrow, the Variables window switches back to the ShellSort display. You can enter any of the ShellSort variable names whose values you wish to see whenever the program is executing in this subroutine.

If you haven't finished executing the program, stop it now using the Stop command.

The debugger is capable of displaying any scalar quantity. Scalar variables include integers, real numbers, strings, pointers, booleans, and characters. Integers, reals, and strings are stored internally in a variety of formats; the debugger can display any of these formats. The debugger can also show values pointed to by a pointer, fields within a record, or elements of an array, so long as the actual thing you are trying to display is ultimately a single value. For example, you can use

```
r.h1
```

to display one of the fields within the rectangle record in the bull’s eye program, although you cannot just type `r` to try to display the entire record.

The rules you use to type complex expressions are covered in detail in Chapter 7, but some simpler rules of thumb are probably all you need. First, array subscripts must always be constant values, not expressions or variables. To look at the value of a pointer, type it's name; to look at the value the pointer points to, type the name followed by ^, just like you would in a Pascal program. You can use `^.` to look at a field in a record that is pointed to by a pointer, just like you do in Pascal. Finally, you can use pointer operators (^), field operators (the `.` character) and array subscripts in combination.

The debugger is case insensitive, just like the Pascal language. Whether you type `MINE` or `mine`, you'll see the same value either way. If you use the star button to fill in the variable values, though, they will always show up in uppercase.

Those Other Disks

There are three other disks in your ORCA/Pascal package that we haven't used yet. Two are labeled "Extras Disk" and "More Extras Disk," these have a number of files that you can use if you install ORCA/Pascal on a hard disk, but that you don't have room for on a single 800K floppy disk. If you are using floppy disks and happen to need one of these files, though, you aren't completely stuck: by moving some of the files that you don't happen to be using off of the program disk, you can make room for the files you do need. The extras disk also has a copy of Apple's installer and installer scripts to help you set up a text-based version of ORCA/Pascal or to install ORCA/Pascal on a hard disk. Installation is covered in Appendix C. Two text files that you should eventually read are also on the extras disk. The first, Tech.Support, gives our address, phone number, and several e-mail addresses that you can use to get in touch with us if you have any questions or problems. The second, Release.Notes, lists changes, additions and corrections to this manual.

The last disk is the "Samples Disk." This disk is chock-full of actual Pascal programs, some of which illustrate useful programming techniques, some of which are used later in Chapter 4 to illustrate the various programming environments on the Apple IIGS, and some of which are just plain fun. If you have a question about how to do something on the Apple IIGS, you might look on the samples disk first – there just might be a program on the samples disk that does exactly what you are trying to do.

Chapter 3 – Compiler Directives

What's a Compiler Directive?

When you learn to write programs in Pascal, most books cover relatively straight-forward text-based programs that can be written using Standard Pascal. You don't need to use compiler directives in such simple programs. Later, as you develop more experience, you start to wish the compiler performed just a little differently. For example, if you are trying to write a classic desk accessory on the Apple IIGS, you need a compiler that will generate a special header.

Compiler directives are instructions to the compiler. They give you a way to tell the compiler to do something in a slightly different way than it normally does. With compiler directives, you can ask ORCA/Pascal to create a desk accessory, generate particular kinds of debug code, or even tell the compiler how to optimize the program. This chapter lists the compiler directives used in ORCA/Pascal, and briefly describes what they are for. While you don't need to be intimately familiar with each of the compiler directives to use the system, it is important that you know that they exist, and basically what they do. That way, you will end up saying to yourself "let's see, to make the compiler do...," rather than "gee, it's too bad the compiler can't..."

How Directives are Coded

Compiler directives under ORCA/Pascal follow a common convention. They look very much like a comment, except that the character right after the opening comment character is a dollar sign ($). More than one directive can be included in a single comment by separating them with commas, or you can put each directive in a separate comment. Most can be used anywhere in the source file.

The following example shows how to save the object module to the file MYPROG and generate a listing. The function of the directives themselves will be explained later – this example is simply to show you the correct format.

```
{$Keep 'MYPROG', List+}
```

A Brief Summary of ORCA/Pascal Compiler Directives

The various compiler directives are explained in detail later in this book. All of the compiler directives are described in Chapter 23, and many are explained in the next chapter, which outlines the various kinds of programs you can write with ORCA/Pascal. The table you see below gives you a brief overview of the compiler directives so you know what is available.

directive	use
Append	The Append directive tells the compiler to open a new source file. It's generally used for small, multi-lingual programs. For example, if the file you append is an assembly language source file containing a few assembly language subroutines, the ORCA system will automatically switch to the ORCA/M assembler to assemble the new file.
CDev	Used to create Control Panel Devices (CDevs). CDevs are the small programs executed by the desktop control panel that ships with Apple's System 6.0 software. Chapter 4 discussed CDevs.
ClassicDesk	Used to create Classic Desk Accessories (CDAs). CDAs are the text programs, like the text control panel, that you can use from virtually all programs. Chapter 4 discusses CDAs in more detail.
Copy	The Copy directive tells the compiler to include all of the source code from another file.
DataBank	When you are using the Apple IIGS toolbox, there are a few cases where you need to define a function that will be called by the toolbox itself. Pascal expects a register called the databank register to be set in a specific way, though, and the toolbox does not set the databank register. This pragma tells ORCA/Pascal to set the databank register properly, something it normally does not need to do.
Debug	ORCA/Pascal generates several kinds of debug code to support the PRIZM and ORCA/Debugger source-level debuggers. Generally, you want to turn all of the debug code on, or turn all of it off, using command line switches or PRIZM check boxes. This directive provides closer control of the debug process.
Dynamic	You can create programs that aren't all in memory at one time. These programs are broken up into more than one piece; each piece is called a segment. The Dynamic directive tells the compiler that the subroutines that follow it should be put in a segment that will be left on the disk until it is needed.
Eject	This directive tells the compiler to send a form feed character to the output device. You can use it to format compiled listings that are being sent to the printer.
Float	ORCA/Pascal generally uses SANE for floating-point calculations, but it can also use the Innovative Systems FPE card. This directive disables certain direct calls to SANE so that code generated for the FPE card runs faster.
ISO	ORCA/Pascal has a lot of extensions to the ISO Pascal Standard. You can use this directive to disable all of the extensions, which will make it a lot more likely that your program will run on another compiler.
Keep	The Keep directive lets you hard code a specific keep name in the source file for your program.
LibPrefix	When you use a uses statement to tell the compiler to include a unit, it looks in the current directory and a special library directory. This

	directive is used to tell the compiler to look somewhere else, like a special, private library.
List	This directive tells the compiler to create a source listing as it compiles the program.
MemoryModel	All 65816 programs must consist of chunks of code that are 64K or smaller, and ORCA/Pascal further assumes that all arrays and records are 64K or smaller. The MemoryModel directive moves all global variables to segments that are not combined with the executable code for the program, so the total size of your global variables can exceed 64K. It also tells the compiler not to assume that arrays and records allocated using dynamic memory are smaller than 64K. The segment statement can also be used to break a program into smaller pieces.
Names	You can use this directive to turn on trace-back code. With this option enabled, if your program stops due to a run-time error, it will print the line that the error occurred on and what subroutines were called to get there.
NBA	Used to create New Button Actions (NBAs) for HyperStudio. NBAs are discussed in Chapter 4.
NewDeskAcc	Used to create New Desk Accessories (NDAs). NDAs are the small programs available under the Apple menu in most desktop programs. NDAs are discussed in Chapter 4.
Optimize	ORCA/Pascal is an optimizing compiler, but optimizations take time. When you are developing a program it's generally best to turn optimizations off so the compiler compiles quicker. Once the program is finished, turn the optimizations on. The compiler will take a lot longer to compile the program, but the program will generally be a lot smaller and faster, too. This directive lets you control the level of optimization.
RangeCheck	The RangeCheck directive turns on some extra checks that lets the compiler check for some kinds of errors. The include using a nil pointer, some kinds of stack overflows, indexing past the end of an array, and setting a value to something outside of the specified subrange.
RTL	ORCA/Pascal programs normally return to the program that launched them using a GS/OS Quit call. Some kinds of programs need to exit with an RTL instruction; the most common example is an Init. This pragma tells the compiler to use an RTL instruction to exit. Inits, and the use of this pragma, are discussed in Chapter 4.
Segment	Programs on the 65816 can be any size, but each piece of the program (called a segment) has to be smaller than 64K. This directive let's you break the program up into more than one segment.
StackSize	Local variables and some information used as functions are called are stored in a special area of memory called the stack. If your program uses too much stack space, it could crash or cause other programs (like

	PRIZM) to crash. By default, your program has 4K of stack space; this pragma is used to increase or decrease the stack space.
Title	The Title directive tells the compiler to print a heading on each page of a compiled listing.
ToolParms	When you are using the Apple IIGS toolbox, there are a few cases where you need to define a function that will be called by the toolbox itself. The toolbox does not return function values the same way Pascal does, though. This directive is used to tell the Pascal compiler to create a function that returns values using the toolbox convention. You will also need to use the databank directive.
XCMD	Used to create XCMDs for HyperCard. XCMDs are discussed in Chapter 4.

Chapter 4 – Programming on the Apple IIGS

The Apple IIGS is a very flexible machine. With it, you can write programs in a traditional text environment, in a high-resolution graphics environment, or in a Macintosh-style desktop environment. ORCA/Pascal lets you write programs for all of these environments, and also supports a number of specialty formats, like desk accessories and HyperCard XCMDs. In this chapter, we will look at each of the programming environments in turn, examining how you use ORCA/Pascal to write programs, what tools and libraries are available, and what your programs can do in each of the environments.

Text Programs

Text programs are by far the easiest kind of programs to write. To write characters to the shell window, you use the appropriate built-in subroutines, like writeln. Input is just as easy – you use built-in subroutines like readln to read characters from the keyboard. Later, when the shell is covered in detail, you will also see that text programs can be executed as a command from the shell window, or even used from the text based programming environment.

As an example, we'll create a simple text program to show how many payments will be needed to pay off a loan for any given interest rate, loan amount, and payment. The variables are placed at the top of the program as constants, so there is no input.

This is actually the first time we have created a program from scratch in this manual, so we will go over the steps involved fairly carefully. If you aren't in the development environment, boot it now. Pull down the File menu and use the New command to open a new program window. Be sure and check the languages menu - PASCAL should be checked. If it is not, select PASCAL from the languages menu. Now type in the program shown below. If you have trouble using the editor, glance through Chapter 7 for help.

(Note: Although the point of this example is to show you how to type in a program from scratch, we should point out that the following example is also on the samples disk in the Text.Samples folder.)

```
{--------------------------------------------}
{                                            }
{  Financing                                 }
{                                            }
{  This program prints the balance on an     }
{  account for monthly payments, along with the }
{  total amount paid so far.                 }
{                                            }
{--------------------------------------------}

program Finance(output);
```

```
const
   loanAmount    = 10000.0;      {amount of the loan}
   payment       = 600.0;        {monthly payment}
   interest      = 15;           {yearly interest (as %)}

var
   balance: real;                {amount left to pay}
   month: integer;               {month number}
   monthlyInterest: real;        {multiplier for interest}
   paid: real;                   {total amount paid}

begin
{set up the initial values}
balance := loanAmount;
paid := 0;
month := 0;
monthlyInterest := 1.0 + interest/1200.0;

{write out the conditions}
writeln ('Payment schedule for a loan of ', loanAmount:10:2);
writeln ('with monthly payments of ', payment:5:2, ' at an');
writeln ('interest rate of ',interest:1,'%.');
writeln;
writeln ('month':15, 'balance':15, 'amount paid':15);
writeln ('-----':15, '-------':15, '-----------':15);

{check for payments that are too small}
if balance*monthlyInterest - balance >= payment then
   writeln ('The payment is too small!')
else
   while balance > 0 do begin
      {add in the interest}
      balance := balance*monthlyInterest;
      {make a payment}
      if balance > payment then begin
         balance := balance-payment;
         paid := paid+payment;
         end {if}
      else begin
         paid := paid+balance;
         balance := 0;
         end; {else}
      {update the month number}
      month := month+1;
      {write the new statistics}
      writeln (month:15, ' ':5, balance:10:2, ' ':5, paid:10:2);
      end; {while}
end.
```

Once the program is typed in, you will need to save it to a work disk. The best choice is a second disk drive, whether that disk drive is a 3.5" drive or a 5.25" drive. Don't be concerned about the amount of disk space available – a 5.25" disk drive has plenty of room for programs. The program disk – that is, the disk with the compiler and linker on it – *must* be in a disk drive when you compile the program, and the disk where you save the program also must be in a drive. The choice of a file name is important, too. Because of the way the ORCA system deals with multi-lingual compiles and partial compiles, and because of some other naming conventions we won't go into now, it's best to pick a name for your program that is ten characters or less, then add .PAS to the name. For this particular program, save it as Finance.pas.

With the program safely on a disk, you are ready to compile it. As with the bull's eye program, you compile the program using Compile to Memory command from the Run menu. If you didn't type the program in properly, an attention box will appear with the error message. When you click OK, you will find the cursor on the exact spot where the error occurred – simply make the correction and recompile. Once the program compiles, it will print the results in the shell window. Unless you shrink the window with your program, you won't see the source window, but the output is still there. You will need to move the shell window and grow it to see all of the results.

One of the classic interactive computer games of all time will serve as our second example, giving us a chance to explore text input and accessing the Apple IIGS toolbox. In this simple game, the computer will pick a distance to a target, and you pick a firing angle for a cannon. The computer then lets you know if you hit the target, or if you missed, by how much. The listing is show below. Go ahead and type it in, but don't compile it yet.

```
{----------------------------------------------}
{                                              }
{  Artillery                                   }
{                                              }
{  This classic interactive text game lets you }
{  pick the angle of your artillery gun in     }
{  an attempt to knock out the enemy position. }
{  The computer picks a secret distance.  When }
{  you fire, you will be told how much you     }
{  missed by, and must fire again.  The object }
{  is to hit the target with the fewest shells. }
{                                              }
{----------------------------------------------}

program Artillery(input, output);

uses Common, MscToolSet;

const
   blastRadius = 50.0;          {max distance from target to get a hit}
   degreesToRadians = 0.01745329; {convert from degrees to radians}
   velocity    = 434.6;         {muzzle velocity}
```

```
var
   angle: real;                 {angle}
   asciiTime: packed array[1..20] of char; {time - for random #s}
   distance: real;              {distance to the target}
   done: boolean;               {is there a hit, yet?}
   time: real;                  {time of flight}
   tries: integer;              {number of shots}
   x: real;                     {distance to impact}
   vx,vy: real;                 {x, y velocities}

begin
{choose a distance to the target}
ReadAsciiTime (@asciiTime);
distance := (ord(asciiTime[16])&$0F*10 + ord(asciiTime[17])&$0F)*100;

{not done yet...}
done := false;
tries := 1;

{shoot 'til we hit it}
repeat
   {get the firing angle}
   write('Firing angle: ');
   readln(angle);

   {compute the muzzle velocity in x, y}
   angle := angle*degreesToRadians;
   vx := cos(angle)*velocity;
   vy := sin(angle)*velocity;

   {find the time of flight}
   {(velocity = acceleration*time, two trips)}
   time := 2.0*vy/32.0;

   {find the distance}
   {(velocity = distance/time)}
   x := vx*time;

   {see what happened...}
   if abs(distance-x) < blastRadius then begin
      done := true;
      write('A hit, after ',tries:1);
      if tries = 1 then
        writeln(' try!')
      else
        writeln(' tries!');
```

```
         case tries of
            1: writeln('(A lucky shot...)');
            2: writeln('Phenomenal shooting!');
            3: writeln('Good shooting.');
            otherwise: writeln('Practice makes perfect - try again.');
            end; {case}
         end {if}
      else if distance > x then
         writeln('You were short by ',round(distance-x):1,' feet.')
      else
         writeln('You were over by ',round(x-distance):1,' feet.');
      tries := tries+1;
   until done;
   end.
```

One of the problems with interactive text programs is that, if you can't see the input, you can't run the program. Before compiling the artillery program, be sure to arrange your windows so you can see the shell window.

By now you've seen that the shell window will open automatically when the program starts to compile, but in a case like this one, you need to open the shell window and resize it before you start to compile the program. There's nothing special about the shell window the system opens for you, so you could just create a new window and change the language type to Shell. You can also open the system's shell window early, though, using the Window menu's Shell Window command.

For the artillery program, you might try leaving the program's window at the full width of the screen, but shortening it so the bottom third of the screen is free. The shell window can be sized to fit in the bottom third of the screen. This arrangement works very well when both the program and it's output use most of the available screen width.

When you run the program, you will see a prompt for the firing angle followed by a black box. This black box is the cursor used by interactive text programs. It lets you know that the input is being read by a program, so normal desktop editing features cannot be used. If you make a mistake, you can use the delete key to back space over your input.

Built-in Procedures and Functions

Your most important resource for writing programs that run in the text environment are the built-in procedures and functions. These are available in any program you write – you do not have to do anything special to get access to them. Built-in procedures and functions include disk access routines like reset, read, and writeln; mathematical functions like cos, arctan, and tan; and many others, like new and dispose. These procedures and functions are described in Chapter 22.

Two points are worth noting about built-in procedures and functions. First, the Standard Pascal requires some procedures and functions – all of these are present in ORCA/Pascal. Some of the required procedures and functions have extended functionality, like the ability to specify a file name when you reset a file. Also, ORCA/Pascal adds many procedures and functions to the list of those required by the ISO standard, like a full set of string manipulation procedures. All of these extensions are flagged as errors if you use the {$ISO+} directive to enforce strict compatibility with the ISO Pascal Standard.

The second point is that you may be porting a Pascal program from another Standard Pascal compiler, and that program may define a procedure or function with the same name as one of the built-in procedures and functions. This does not cause a problem – the local definition will have precedence over the predefined one, so the program will work as expected.

Console Control Codes

When you are writing text programs that will execute on a text screen, one of the things you should know about are the console control codes. These are special characters that, when written to the standard text output device, cause specific actions to be taken. Using console control codes, you can beep the speaker, move the cursor, or even turn the cursor off. The console control codes are covered in Appendix E.

Keep in mind that these console control codes only work with the text screen. While you can write text programs and execute them from the desktop, you cannot use these console control codes to control the output in the shell window.

Stand-Alone Programs

So far, all of the programs you have created have an executable file type of EXE. EXE files are special in the sense that the program environment knows it does not have to shut itself down to run the program. EXE files can also have embedded debug code, and do not have to start the tools for themselves. Unfortunately, they cannot be executed from the Finder.

There are two changes you need to make before any of the text programs you have created so far can run from the Finder. The first is to turn off debug code, which you can do by disabling the "Generate debug code" check box in the Compile dialog. The other change you must make is to change the file type to S16 in the Link dialog; you do this by selecting the S16 radio button. In general, you should also turn off the "Execute after linking" option in the Link dialog, since it's a pretty slow process to run an S16 program directly from PRIZM.

With these changes made, recompile one of your text programs and leave the ORCA environment. From the Finder, you will now see the hand-in-a-diamond program icon, which tells you that you can run the program from the Finder.

Graphics Programs

A large subset of programs need to display graphics information of some kind, but aren't necessarily worth the effort of writing a complete desktop program. These include simple fractal programs, programs to display graphs, slide show programs, and so forth. In this book, these programs are called graphics programs.

Your First Graphics Program

Writing a graphics program with ORCA/Pascal is really quite easy. In general, all you have to do is issue QuickDraw II commands, and be sure the Graphics window is positioned properly before you run your program. QuickDraw II is the largest and most commonly used tool in the Apple IIGS toolbox, so it's also a good place to get started along the road to writing desktop programs.

To learn about QuickDraw II, you will need a copy of the Apple IIGS Toolbox Reference , Volume 2. This book was written by Apple Computer, and is published by Addison Wesley. While the toolbox reference manual is a reference, and thus not an easy book to read, it is essential that you have a copy to answer your specific questions about the toolbox. This section shows a couple of examples so you know how to create graphics programs using ORCA/Pascal, but there is a lot more to QuickDraw II than you see here.

To get access to QuickDraw II, you must include the statement

```
uses Common, QuickDrawII;
```

in your program. You may have noticed this statement in the first sample we ran, the bull's eye program. The bull's eye program also showed how to use the QuickDraw II command DrawOval to draw ovals on the screen. Our next QuickDraw II sample, which draws spirals on the graphics screen, shows the commands MoveTo, which initializes the place where QuickDraw II will start drawing from (called the pen location), and LineTo, which draws a line from the current pen location to the specified spot, moving the pen location in the process.

```
program spiral;

uses Common, QuickDrawII;

var
   r, theta: real;

begin
theta := 0.0;
r := 40.0;
SetPenSize(2, 1);
MoveTo(280, 40);
while r > 0.0 do begin
   theta := theta + 3.1415926535/20.0;
   LineTo(round(cos(theta)*r*3) + 160, round(sin(theta)*r) + 40);
   r := r - 0.15;
   end; {while}
end.
```

Save the program as Spiral.pas. As with the bull's eye program, reduce the width of your source code window to about half the screen width and open the graphics window before executing the program.

Stand-Alone Programs

Any program that uses any of the Apple IIGS toolbox must initialize the tools it uses. ORCA/Pascal automatically initializes several tools, and opens the .CONSOLE device used for text input and output. Graphics programs, though, are using QuickDraw II, and ORCA/Pascal does not automatically start this tool. Before you can run a graphics program from outside of PRIZM, you will have to learn to start and shut down QuickDraw II.

In the case of simple graphics programs, the easiest way to start QuickDraw II is to use the built-in procedure StartGraph. StartGraph uses a single integer parameter to determine the size of screen to use. This parameter should be either a 320 or a 640. If you want the program to produce pictures that look the same as they do in the graphics window, use 640. At the end of your program, you also need to use the built-in procedure EndGraph to shut down the graphics environment. The spiral program is shown below, changed to meet these requirements. The changes are shown in bold-face.

```
program spiral;

uses Common, QuickDrawII;

var
   r, theta: real;

begin
StartGraph(640);
theta := 0.0;
r := 40.0;
SetPenSize(2, 1);
MoveTo(280, 40);
while r > 0.0 do begin
   theta := theta + 3.1415926535/20.0;
   LineTo(round(cos(theta)*r*3) + 160, round(sin(theta)*r) + 40);
   r := r - 0.15;
   end; {while}
EndGraph;
end.
```

You can still run this program from the desktop development environment. The only change you will see is that the menu bar will vanish while the program is executing. This happens any time you start a tool; the system is allowing your program to draw its own menu bar. To switch back to the debugger's menu bar while your program is running, click on the double-arrow icon that appears at the right-hand side of your menu bar.

As with a stand-alone text program, you must remember to turn off debug code and to change the file type to S16. With these changes in place, you can compile the program, creating an executable file that will run from the Finder.

Programming on the Desktop

Most people we talk to want to write programs that use Apple's desktop interface. These programs are the ones with menu bars, multiple windows, and the friendly user interface popularized by the Macintosh computer. If you fall into that group of people, this section will help you get started. Before diving in, though, we want to let you know what you will need to do to write this kind of program.

Anyone who tells you that writing desktop programs is easy, or can be learned by reading a few short paragraphs, or even a chapter or two of a book is probably a descendent of someone who sold snake oil to your grandmother to cure her arthritis. It just isn't so. Learning the Apple IIGS toolbox well enough to write commercial-quality programs is every bit as hard as learning a new programming language. In effect, that's exactly what you will be doing. The Apple IIGS Toolbox Reference Manuals come in four large volumes. Most of the pages are devoted to brief descriptions of the tool calls – about one call per page. It takes time to learn about all of those calls. Fortunately, you don't have to know about each and every call to write desktop programs.

Learning the Toolbox

As we mentioned, learning to write desktop programs takes about the same amount of time and effort as learning to program in Pascal. If you don't already know how to program in Pascal, *learn Pascal first!* Concentrate on text and graphics programs until you have mastered the language, and only then move on to desktop programming.

This doesn't mean that you need to know everything there is to know about Pascal, but you should feel comfortable writing programs that are a few hundred lines long, and you should understand how to use records and pointers, since the toolbox makes heavy use of these features.

The toolbox itself is very large. The Apple IIGS Toolbox Reference Manual is a three volume set that is basically a catalog of the hundreds of tool calls available to you. These three volumes cover the tools up through System 5.0; the additions in System 6.0 are covered in Programmer's Reference for System 6.0. This four-volume set is an essential reference when you are writing your own toolbox programs. A lot of people have tried to write toolbox programs without these manuals. I can't name a single one that succeeded.

A lot of people have been critical of the toolbox reference manuals because they do not teach you to write toolbox programs, but that's a lot like being critical of the Oxford English Dictionary because it doesn't teach you to write a book. The toolbox reference manuals are a detailed, technical description of the toolbox, not a course teaching you how to use the tools. Toolbox Programming in Pascal does teach you the toolbox, though. This self-paced course also includes an abridged toolbox reference manual, so you can learn to use the toolbox before you spend a lot of money buying the four volume toolbox reference manual.

All of this is not meant to frighten you away. Anyone who can learn a programming language can learn to write desktop programs. Unfortunately, too many people approach desktop programming with the attitude, fostered by some books and magazine articles, that they can learn to write desktop programs in an evening, or at most a weekend. This leads to frustration and usually failure. If you approach desktop programming knowing it will take some time, but willing to invest that time, you *will* succeed.

Hardware Requirements

Toolbox programs are big, and they bring in over a dozen very large interface files, too. While ORCA/Pascal can be used to write programs on a fairly small Apple IIGS, you cannot write toolbox programs with the minimal system.

The first thing you need is more disk space. To write toolbox programs, you will need access to more interface files than will fit on a floppy disk with the rest of the ORCA system. You will also need to use Apple's Rez resource compiler, which is fairly large in it's own right. Because of the amount of disk space you need, the only practical solution is a hard disk.

The other thing you will need to write any large program is more than 1.25M of memory. You can squeak by with 1.75M of memory, although you may have to reboot fairly often, break your program up into lots of pieces, or even switch to the text environment. We recommend 2M or more of memory for toolbox programming.

Toolbox Interface Files

As you look through the toolbox reference manuals, you will see that the toolbox is divided into a set of tools, each with its own name. There is a unit for each of these tools; it contains definitions for all of the tool calls and data structures used by the tools. In some cases, a data structure might be shared by more than one tool; in that case, the definition is in the unit Common, which is needed for almost all of the tool units. That's why you always see Common listed before QuickDrawII, even in the short graphics programs.

There are actually two different versions of the tool units. One version, which you can find on the More Extras disk at the path :MoreExtras:Apple:Libraries:AppleTools, was created by Apple Computer for the MPW IIGS Pascal compiler. We've made sure these units will compile under ORCA/Pascal, and offer them for people who might be trying to move programs that were developed with these interfaces, or who need to develop programs under both ORCA/Pascal and MPW. In this manual, and in our courses, though, we will use the Byte Works interfaces.

Here's a list of the current toolbox files and the tool they define. The name shown is the name you would use in the uses statement; on disk, the file will have .int appended, and be located at 13:ORCAPascalDefs. A few, like GSOS and FINDER, don't technically document the tools, but they are included here for completeness.

Uses Name	Tool
ACE	Audio Compression/Expansion
Common	types and constants
ControlMgr	Control Manager
DeskMgr	Desk Manager
DesktopBus	Apple Desktop Bus

DialogMgr	Dialog Manager
EventMgr	Event Manager
Finder	Finder Interface
FontMgr	Font Manager
GSOS	GS/OS Disk File Manager
HyperStudio	HyperStudio Interface
HyperXCMD	HyperCard Interface
IntegerMath	Integer Math Tool Set
LineEdit	Line Edit Tool Set
ListMgr	List Manager
MemoryMgr	Memory Manager
MenuMgr	Menu Manager
MIDI	MIDI Sound Tools
MIDISynth	MIDISynth Tool Set
MscToolSet	Miscellaneous Tool Set
MultiMedia	Media Control Tool Set
Sequencer	Note Sequencer
Synthesizer	Note Synthesizer
ORCAShell	ORCA Shell Interface
PrintMgr	Print Manager
PRODOS	ProDOS Disk File Manager
QuickDrawII	QuickDraw II and QuickDraw Auxiliary
ResourceMgr	Resource Manager
Scheduler	Scheduler
ScrapMgr	Scrap Manager
SFToolSet	Standard File Operations Tool Set
SoundMgr	Sound Manager
TextEdit	Text Edit Tool Set
TextToolSet	Text Tool Set
ToolLocator	Tool Locator
WindowMgr	Window Manager

Table 4.1: Summary of Interface Tool Files

Debugging a Desktop Program

Debugging a desktop program is not much more difficult than debugging a text or graphics program, but there are a few points you need to keep in mind. These arise from the fact that both the debugger and your program need the mouse, keyboard, and menu bar to function.

As soon as the debugger decides that your program is a desktop program, you will see your menu bar replace the desktop menu bar. The debugger makes this decision based on tool startup calls. If you initialize any tool in Table 4.1 except SANE, the debugger treats your program like a desktop program. ORCA's windows are still visible, but you can no longer select them. At the far right of your menu bar, you will see two special icons, created by the debugger. The first is a footprint and the second is a combined left and right arrow. The footprint is used to step through your program, one line at a time, without having to return to the desktop. The arrows are used to return to the desktop to issue some other debugging command. If you switch to the desktop while you are debugging your program, you will see that the special icons are also in the Desktop's menu bar. You can select the arrows icon to return to your program.

You *should not* switch menu bars while your program is creating its menu bar. From the time you issue the first Insert Menu tool call until you draw the menu bar, your menu bar is incomplete. This restriction should not pose any special problems if you are building standard menus, but could be troublesome in the case where you are defining your own menus. To debug your menu bar routine, then, you will need to limit your debugging activities to clicking on the step icon.

A second source of potential trouble lies in trying to debug your window update routine. Again, you should not switch to the desktop during your update routine, since the debugger might need to use your routine to repaint your windows. You should use the footprint icon to invoke the Step command to debug your update routine.

A third problem area regards your program stack. The debugger will be using your stack, so you need to be sure that you do not use coding tricks that depend on the values below the stack pointer remaining unchanged. The Pascal compiler doesn't do this; it would only happen in your own assembly language subroutines. You also need to make sure that there are at least 256 bytes of free stack space at all times.

The fourth point is that you should not issue a Stop command in the middle of debugging, but instead let your program continue to execute until it reaches its natural conclusion. This restriction applies to the case where you have started tools that were not started by the desktop, and a premature abort from your program will leave these tools open. It is assumed that your program shuts down any tools it starts; the debugger looks over your shoulder and prevents startup calls for tools already initialized, and also prevents shutting down tools it needs. The debugger does not shut down any extra tools you have initialized. The desktop starts up the following tools:

- Control Manager
- Desk Manager
- Dialog Manager
- Event Manager
- Font Manager
- Line Edit Tool
- List Manager
- Memory Manager
- Menu Manager
- Print Manager
- QuickDraw Auxiliary
- QuickDraw II
- SANE
- Scrap Manager
- Standard File Manager

- Tool Locator
- Window Manager

Table 4.2 – Tools started by ORCA/Desktop

Keep in mind that since these tools are already active when your program executes, debugging may not reveal errors associated with failure to load and start these tools.

A fifth area of trouble is switching to the desktop between paired events in your program. For example, the code which handles mouse-down events and mouse-up events is usually closely connected. A switch to the debugger causes a flush of the event queue. If you switch to the desktop after detecting one kind of event, then return to your program where you await that event's paired ending, your program may go into a state of suspended animation. You can avoid this problem by carefully considering where switches to the desktop are not dangerous. Don't switch menu bars if you are in doubt!

There are two restrictions on the kind of desktop programs you can debug. The desktop handles 640 mode only; you should use 640 mode while you are debugging your program. The second is that the file type of your program can only be EXE or NDA (GS/OS executable file or new desk accessory, respectively). You should change your program's file type to one of these during debugging, and then change it back to whatever you want after you have the program running.

Writing New Desk Accessories

New desk accessories are those programs which can be selected from the apple menu of a desktop program. The principal advantage of a desk accessory is that it can be used from any desktop program which follows Apple's guidelines. Writing a desk accessory is not hard, but it does require the compiler to generate special code, so you must write a desk accessory in a special way. For the most part, though, writing a desk accessory uses the same tools and techniques you use to write desktop programs.

Your desk accessory starts with the nda directive. This directive has seven parameters. The first four are the names of four subroutines in your program that have special meaning in a desk accessory. The next two are the update period and event mask. The last is the name of your desk accessory, as it will appear in the Apple menu. The format is:

`{$NewDeskAcc` *`open close action init period eventMask menuLine`*`}`

open	This parameter is an identifier that specifies the name of the function that is called when someone selects your desk accessory from the Apple Menu. It must return a pointer to the window that it opens.
close	This parameter is an identifier that specifies the name of the procedure to call when the user wants to close your desk accessory. It must be possible to call this function even if open has not been called.

action	The action parameter is the name of a procedure that is called whenever the desk accessory must perform some action. It must declare two parameters. The first is an integer parameter, which describes the action to be taken. The second parameter is a pointer to an event record. See page 5-7 of the Apple IIGS Toolbox Reference Manual for a list of the actions that will result in a call to this function.
init	The init parameter is the name of a procedure that is called at start up and shut down time. This gives your desk accessory a chance to do time consuming start up tasks or to shut down any tools it initialized. This procedure must define a single integer parameter. The parameter will be zero for a shut down call, and non-zero for a start up call.
period	This parameter tells the Desk Manager how often it should call your desk accessory for routine updates, such as changing the time on a clock desk accessory. A value of -1 tells the Desk Manager to call you only if there is a reason, like a mouse down in your window; 0 indicates that you should be called as often as possible; and any other value tells how many 60ths of a second to wait between calls.
eventMask	This value tells the Desk Manager which events your desk accessory can handle. You are only called for events allowed by this mask. It works the same way as the event masks we used in the Frame sample program.
menuLine	The last parameter is a string. It tells the Desk Manager the name of your desk accessory. The name must be preceded by two spaces. After the name, you should always include the characters \H**.

While the compiler will still expect to see the body of a program, it will never be executed, so there is no need to put anything there. That is, you should code a `begin end.` block but you aren't required to place any statements between them.

The format for a sample desk accessory, then, is:

```
{$NewDeskAcc StartUp ShutDown Action Init 60 1023 '  Sample\H**'}
program SampleDesk;

uses Common, WindowMgr;

function Open: grafPortPtr;

begin
<<<open the window and assign the pointer>>>
end;
```

```
procedure ShutDown;

begin
<<<close the window>>>
end;


procedure Action (event: integer; param: eventRecord);

begin
<<<handle events>>>
end;


procedure Init (code: integer);

begin
if code = 0 then
   <<<shutdown code>>>
else
   <<<startup code>>>
end;


begin
{nothing goes in the program body}
end.
```

Once you have written a desk accessory, you must install it. For the desk manager to find your desk accessory, it must be located on the boot volume in a directory called SYSTEM:DESK.ACCS. It also has a special file type, called NDA. To create the desk accessory, select the NDA file type from the Link dialog that appears when you use the Link command from the Run menu. Be sure and turn debugging off for your final compile!

For a sample desk accessory that illustrates these principles, see Clock.pas on the samples disk.

Debugging NDAs

Normally, to run a new desk accessory, you install it in the desk accessories folder. From that time on, the desk accessory is available to any desktop program that supports desk accessories. When you are developing a desk accessory, though, you don't want to reboot every time you change the program. Instead, the desktop development environment allows you to execute a new desk accessory just like any other program. Be sure and use the Link dialog box in the Run menu to change the file type of the file to NDA, though. If the file type is not set to NDA, the desktop development environment does not know that the file is a desk accessory, and the program will almost certainly crash when you try to execute it. When you are developing the desk accessory, you do not have to move it to the desk accessories folder, nor do you have to execute it from the

Apple menu. You can also leave the debug code turned on, and debug the desk accessory just like any other desktop program. When you execute the desk accessory this way, the development environment simulates the same conditions that the desk accessory will face when it is executed from the Apple menu.

Once the program is finished, you can turn off debugging and move the program to the desk accessories folder.

Classic Desk Accessories

Classic desk accessories are the utility programs that you can run by holding down the and control keys while you press the esc key. Classic desk accessories are really just a special form of text programs. Like text programs, they use the text screen for input and output. The advantage of a desk accessory is that it can be used from virtually any program, even programs like the text version of AppleWorks that don't even know they exist. (If you want to create a desk accessory that will be used from ProDOS 8 programs, though, you must not use Pascal's standard disk I/O routines.)

Classic desk accessories have a special header. You tell the compiler to create a classic desk accessory using the ClassicDesk directive. This directive has a series of three parameters which tell the compiler how to build the special header required by the Apple IIGS. These parameters, in the order in which they appear in the directive, are:

name	A string giving the name of the desk accessory. This name will appear in the menu when you press -control-esc.
start	Start is the procedure that is called when your program is selected from the list of available classic desk accessories. The start procedure serves the same purpose as the program body in a normal Pascal program.
shut down	Shut down is a procedure that is called when a program stops, or just before the Apple IIGS switches operating systems. It gives your program a chance to clean up after itself, in case your program started some background task.

The echo program from the CDA.SAMPLES folder of the samples disk shows a very simple classic desk accessory.

```
{$ClassicDesk 'Echo from Pascal' Start ShutDown}
{---------------------------------------------------------------}
{                                                               }
{  Echo                                                         }
{                                                               }
{  This is about the simplest a classic desk accessory can be,  }
{  providing a quick framework for developing your own.  It     }
{  simply reads strings typed from the keyboard and echos       }
{  them back to the screen.                                     }
{                                                               }
{  Mike Westerfield                                             }
{                                                               }
{  Copyright 1987-1988                                          }
{  Byte Works, Inc.                                             }
{                                                               }
{---------------------------------------------------------------}

program Echo (input, output);

procedure Start;

var
   str: string[255];

begin
writeln ('This program echoes the strings you type from the keyboard.');
writeln ('To quit, hit the RETURN key at the beginning of a line.');
writeln;
repeat
  readln (str);                       { read a string }
  writeln (str);                      { write the same string }
until length (str) = 0;               { quit if the string is empty }
end;


procedure ShutDown;

begin
end;

begin
{main program - nothing goes here}
end.
```

As you can see, the directive should appear right at the top of the program. The Start and ShutDown parameters are the actual names of the subroutines that will be called; you can use any legal Pascal function names you like.

Once you write a classic desk accessory, there are several things you must do to make it available to your programs. First, be sure to turn off the debug flag, so no debug code is generated. If you miss this step, your program will crash when you try to execute it. Next, bring up the Link dialog using the Link command from the Run menu. You will see a series of four buttons, one for each file type supported by the desktop development environment. One of these, CDA, tells the system that the executable file type must be CDA. Select this button. In addition, deselect the "Execute after link" option; you cannot execute a classic desk accessory directly from the desktop. At this point, you are ready to compile the program. Once the program is compiled, copy it to the DESK.ACCS folder of the SYSTEM directory. The next time you boot your computer, the classic desk accessory will appear in the desk accessories menu.

Debugging Classic Desk Accessories

Since classic desk accessories use the text screen, they cannot be debugged from the desktop debugger. You can however, make a very slight change to any classic desk accessory, and debug the resulting "normal" program. To do this, turn the {$ClassicDesk} directive into a true comment by removing the $ character, and then add two statements to your program body, one to call the Start procedure and a second to call the ShutDown procedure. The program body for the echo sample, shown above, would be

```
begin
Start;
ShutDown;
end.
```

After the program is debugged, remove the two calls, replace the $ character in the directive, and proceed with installing the desk accessory as you normally would.

Inits

Initialization programs are a special kind of program that is executed as your computer boots. There are a number of special requirements for Inits, but only two effect the way you use ORCA/Pascal.

When most Pascal programs are complete, ORCA/Pascal makes sure a GS/OS Quit call is executed; this shuts down the program and returns control to the Finder (or whatever program launcher was used). Initialization programs must exit with an RTL instruction, instead. To accomplish this, place an rtl directive at the start of the program. The rtl directive has no parameters. The rest of the program looks just like a normal Pascal program.

The other special requirement is to set the file type to either PIF (for a permanent initialization program) or TIF (for a temporary initialization program). In practice, you will also need to use the Rez compiler to create the icon that shows up when the program starts. Since this means you have at least three steps – compiling the Pascal program, compiling the resources, and setting the file type – in practice initialization files are almost always built with script files.

For an example of a very simple TIF, see the Samples disk. For more information about the shell and script files, see Chapter 6 and Chapter 8. The resource compiler is covered in Chapter

10. For information about writing initialization programs that is not ORCA/Pascal-specific, see the Apple IIGS File Type Notes for file types $B6 (PIF) and $B7 (TIF).

HyperStudio NBAs

You can write HyperStudio New Button Actions (NBAs) with ORCA/Pascal with the aid of the NBA directive and the HyperStudio interface file. The format for the NBA directive is:

```
{$nba main}
```

main	This parameter is an identifier that specifies the name of the procedure that is called when HyperStudio calls the NBA. This procedure accepts one parameter of type HSParamPtr.

You still need a program body, but it should not have any statements.

The parameter that is passed to the function is a pointer to the HyperStudio parameters, contained in a record. The structure itself is defined in the HyperStudio interface file.

HyperStudio supports a wide range of callbacks, which are calls you can make from the NBA back to HyperStudio to perform some action. These are called via the __NBACALLBACK procedure, which takes two parameters. (Note: there are two _ characters at the start of the name, not one!) The first is the callback number, while the second parameter is a pointer to a parameter record.

You will find a constant defined for each of the callbacks in the file HyperStudio.pas, which is the source file used to create the HyperStudio interface file. Roger Wagner Publishing distributes technical information about the callbacks themselves.

The parameter record can be, and usually is, the same one that is passed to the NBA when it is called. If you decide to create copies of the parameter structure, be sure you actually copy the original structure into the copy. There are several internal fields, notably the callback address, which must be set before you make the callback.

For an example of a simple NBA, see the Samples disk.

HyperCard XCMDs

You can write HyperCard XCMDs and XCFNs with ORCA/Pascal with the aid of the xcmd directive and the HyperXCMD interface file. The format for the xcmd directive is:

```
{$xcmd main}
```

main	This parameter is an identifier that specifies the name of the procedure that is called when HyperCard calls the XCMD. This procedure accepts one parameter of type XCMDPtr.

You still need a program body, but it should not have any statements.

The parameter that is passed to the procedure is a pointer to the HyperCard parameters, contained in a record. The structure itself is defined in HyperXCMD interface file.

HyperCard supports a wide range of callbacks, which are calls you can make from the XCMD back to HyperCard to perform some action. HyperCard callbacks work like tool calls, although technically they have a unique entry point, and are not tools. HyperXCMD has a complete set of function declarations for the HyperCard callbacks.

HyperCard XCMDs and XCFNs are documented on the System 6.0 CD ROM.

For an example of a simple XCMD, see the Samples disk.

Control Panel Devices (CDevs)

You can write Control Panel CDevs with ORCA/Pascal with the aid of the cdev directive. The format for the cdev directive is:

```
{$cdev main}
```

main	This parameter is an identifier that specifies the name of the function that is called when the Control Panel calls the CDev. This function accepts an integer parameter and two long integer parameters, in that order, and returns a long integer. The parameters and return value are explained in the references mentioned below.

For a description of the parameters and the value returned by the function, along with the other information you need to write CDevs, see the Apple IIGS File Type Notes for file type $C7 (CDV).

For an example of a simple CDev, see the Samples disk.

Chapter 5 – Writing Assembly Language Subroutines

Introduction

By using the ORCA/M macro assembler with ORCA/Pascal, it is easy to write assembly language subroutines that can be called from Pascal programs. This chapter describes in detail how this is done. You do not need to know the information in this chapter to write Pascal programs. To understand all of the information in this chapter, you must already know assembly language and how to use ORCA/M or APW. Before continuing, you should install the ORCA/M or APW assembler in your Pascal development system.

The Basics

Calling an assembly language subroutine from Pascal is actually quite easy. For our first example, we will take the simplest case: a procedure defined in assembly language that has no parameters and does not use any global variables from Pascal.

The first step is to tell the Pascal compiler that the procedure will not appear in the program, but that it will, in fact, be found by the linker. One of the important things to remember about this is that when you start dealing with procedures and functions defined outside of the Pascal part of the program, you will always have to tell the compiler what you are doing. The compiler is a trusting program - it will always believe you. The compiler can check the Pascal program to make sure that it agrees with your statements about what is defined elsewhere, but it is up to you to make sure that you tell the compiler the truth. If you lie, on purpose or by accident, the results can be dramatic! So be careful when you tell the compiler about the procedure.

We will define a small procedure to clear the keyboard strobe. This is one of those tasks that is difficult to do from Pascal, yet takes only four lines of assembly language. You might want to call this procedure from a real program - the effect is to erase any character that the user has typed, but that has not yet been processed by a read statement.

The Pascal program must declare the procedure as `extern`. This is how you tell the compiler that the procedure appears outside of the Pascal part of the program. A program that simply calls the procedure would look like this:

```
program ClearStrobe;

procedure Clear; extern;

begin
Clear;
end.

{$append 'myprog.asm'}
```

Once you have typed the program in, save it as `MYPROG.PAS`. Be sure the file type is Pascal. You can check this by pulling down the languages menu.

The append directive at the end of the program is appending assembly language code to the end of a Pascal program. The compiler is smart enough to look ahead at the language type stamped on the file being appended. If it is an assembly language file, the compiler calls the assembler to process it. This is one of the things that makes multi-lingual programming so easy with the ORCA development system. If you are more familiar with separate compilation, you can, of course, use that method.

At this point we need to add the assembly language procedure. Create a new window, then pull down the Languages menu and select ASM65816 to change the language stamp of the window to assembly language. With that accomplished, type in the procedure shown below.

```
;
;  Clear the keyboard strobe
;
Clear      start
           sep      #$20
           sta      >$C010
           rep      #$20
           rtl
           end
```

Save the file as `MYPROG.ASM`, the same name that appeared in the append directive at the end of the Pascal program.

Now for the fun part. Select the MYPROG.PAS window, and then use the Compile to Disk command, as if the program is written entirely in Pascal. It doesn't matter if the assembly language source file is open on the desktop or not.

△ **Important** Be sure to use the Compile to Disk command, not the Compile to Memory command or one of the debug commands. You must compile to disk any time more than one source file is needed to create a program. △

What happens is this:

1. ORCA looks at the file `MYPROG.PAS`. Since it is a Pascal file, the Pascal compiler is called to compile the program.

2. When the compiler gets to the append directive, it looks at the file. Since it is not a Pascal program, control is returned to the desktop development environment.

3. The development environment sees that there are more source files to process. The file is an assembly language file, so the assembler is called. The assembler assembles the subroutine.

4. The linker is called. It links the Pascal and assembly language parts into one program and writes an executable file called `MYPROG`.

5. The program is executed.

Returning Function Values From Assembly Language Subroutines

Function values are returned in the registers. This means that within your assembly language subroutine you would load the registers with the values that you want to return to your Pascal program. Boolean, character, and integer values are returned in the accumulator as two-byte quantities. Long integers and pointers are returned in the X and A registers, with the most significant word in X and the least significant word in A. Real numbers are returned as pointers to floating-point values which have been stored in SANE's extended format. This format is described in Apple Numerics Manual. As with other types of pointers, the most significant word should be placed in X and the least significant word should be stored in A.

Please note that characters and boolean values only require one byte of storage, but are returned in a two-byte register. Be sure to zero the most significant byte of the value that you return.

For a complete discussion of the internal formats of numbers, see Chapter 12. Basically, though, they correspond to what you are used to in assembly language.

Our next example program illustrates how to implement an assembly language function from Pascal. The Pascal program stays in a tight loop, repeatedly calling an assembly language subroutine, named KEYPRESS to see if a key has been pressed. Once a key has been pressed, it calls another assembly language subroutine, named CLEAR, to clear the strobe.

```
program Wait;

procedure Clear; extern;

function Keypress: boolean; extern;

begin
while not Keypress do {nothing};
Clear;
end.

{$append 'myprog.asm'}
```

Once this file is entered, check to be sure its language stamp is Pascal, and save it as MYPROG.PAS. Next, type in the following assembly language file, make sure it is stamped as ASM65816, and save it as MYPROG.ASM.

```
;
;  Return the status of the keyboard strobe
;
Keypress   start
           sep      #$20
           lda      >$C000              get keyboard key
           asl      A                   roll high bit to A
           rep      #$20
           lda      #0
           rol      A
           rtl
           end

;
;  Clear the keyboard strobe
;
Clear      start
           sep      #$20
           sta      >$C010
           rep      #$20
           rtl
           end
```

Passing Parameters to Assembly Language Subroutines

To better understand the interaction between Pascal and assembly language in the ORCA environment, we will look at how parameters are passed from a Pascal program to an assembly language subroutine. ORCA/Pascal places the parameters which appear in a subroutine call on the stack in the order that they appear in the parameter list. It then issues a JSL to your subroutine.

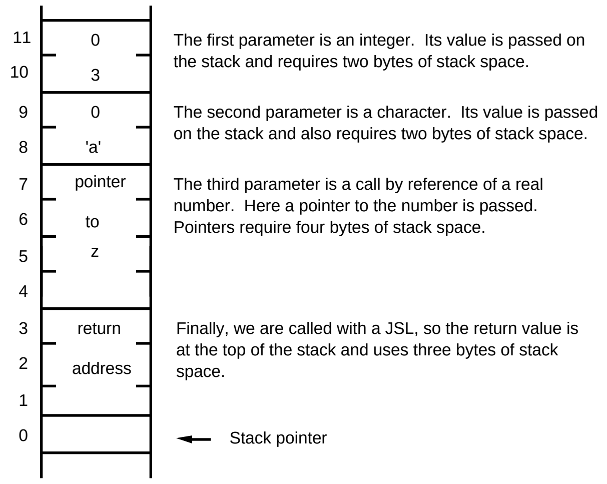
The value that is on the stack depends partly on the type of the value being passed and partly on the way the procedure or function header is declared. Value parameters, including integers, booleans, sets and characters all appear on the stack as actual values, using the same format that is used to store them in main memory. Real, double, extended and comp values are always converted to extended before being passed, so they always appear on the stack as ten-byte values. Subranges of these values or enumerations also appear as actual values. Any other parameter is passed as an address that points to the first byte of the value. This includes structured value parameters, like arrays or records, as well as var parameters. When a procedure or function is passed as a parameter, its address is placed on top of the stack.

Notice that a structured type, like an array, is passed as an address even if the procedure or function header says it is a value parameter. While there is nothing to stop you from changing the values in the structured type, it is confusing to do so. If you will be changing the values of something passed as a parameter, declare the parameter as a var parameter in the Pascal program.

Consider the Pascal program fragment below:

```
...

procedure DoSomething (i: integer; ch: char; var z: real); extern;

...

i := 3;
ch := 'a';
z := 5.6;
DoSomething (i, ch, z);

...
```

When DoSomething is called, the stack will look like this:


11
10
9
8
7
6
5
4
3
2
1
0
0
3
0
'a'
pointer
to
z
return
address
The first parameter is an integer. Its value is passed on the stack and requires two bytes of stack space.
The second parameter is a character. Its value is passed on the stack and also requires two bytes of stack space.
The third parameter is a call by reference of a real number. Here a pointer to the number is passed. Pointers require four bytes of stack space.
Finally, we are called with a JSL, so the return value is at the top of the stack and uses three bytes of stack space.
Stack pointer

In order to access the passed parameters in our assembly language subroutine, we first need to set up a local direct page, using the stack. *Be very careful to save and restore the direct page register! Upon entry to the subroutine, we do not know where the direct page register points – failure to restore it could lead to disastrous results!*

One of the simplest ways to set the direct page register equal to the stack pointer is to transfer the stack register contents to the accumulator, save the current direct page register by pushing it onto the stack, and then set the new direct page register by transferring the contents of the accumulator to the direct page register:

```
tsc
phd
tcd
```

Before leaving the subroutine, we can restore the old value of the direct page register by pulling it from the stack:

```
                pld
```

We are now in a position to access the passed parameters as direct page locations. Referring to the stack diagram given above, we can code a series of equates, setting the positions in the stack to local labels:

```
i           equ       10
ch          equ       8
z           equ       4
```

After setting up a direct page from the stack, `i` and `ch` can now be accessed as simple direct page values, as in

```
            lda       i
            lda       ch
```

while `z`, since it is a pointer, requires long indirect addressing:

```
            lda       [z]
```

Since we know the location of the return address (it is at direct page location 1 in our example), we can pop the parameters from the stack by moving the return address to the last three bytes of the parameter area and then removing bytes from the stack. Putting this all together, the Pascal program below shows how to implement an assembly language function. The program does little more than define an integer and then call a function to reverse the bits in the integer. If you are not sure how the assembly language program works, make yourself a stack diagram. If this still seems like magic, do not lose heart. The next section covers two assembly-language macros provided with ORCA/Pascal to alleviate the burden of manipulating the stack frame from assembly language.

```
{ Demonstrate calling assembly language functions from Pascal. }

program FunctionIt (output);

function reverse (a: integer): integer; extern;

begin
writeln (6, reverse(6));
end.

{$append 'reverse.asm'}
```

```
;
;  Reverse the bits in an integer
;
Reverse    start
parm       equ      4       passed parameter
ret        equ      1       return address

           tsc              record current stack pointer
           phd              save old DP
           tcd              set new DP to stack pointer

           ldx      #16     place result in A
lb1        asl      parm
           ror      A
           dex
           bne      lb1
           tax              save result in X
           lda      ret+1   set up stack for return from
           sta      parm     subroutine
           lda      ret-1
           sta      ret+1
           pld              restore old DP
           pla              set stack ptr for return
           txa              put the function result in A
           rtl
           end
```

The Macro Solution

For most purposes, dealing directly with the stack to examine and remove parameters, return function values, and so on, can be tedious and error prone. To make things easier, there are two macros on the distribution disk that help you in passing parameters, returning function values, and setting up direct page work space. The macros are called SUBROUTINE and RETURN, and are located in a file called MACROS on the SAMPLES disk.

SUBROUTINE is designed to be used right after the START directive of an assembly language subroutine. It has two operands: the first is a parameter list, and the second is the number of bytes of direct page work space you want for your own use. If your assembly language subroutine is expecting more than one parameter, then you need to enclose the parameter list in parentheses. Each of the parameters starts with a number indicating how long the parameter is in bytes, and is followed by a colon and the name of the parameter. The parameters are specified in the same order in which they are given in the call statement of the Pascal procedure or function. The work space parameter is a number, specifying the number of bytes of work space you need. The work space starts at direct page location zero.

For example, let's assume that you have defined an assembly language procedure named Sample, with pass parameters as indicated below.

```
procedure Sample (var i: integer; r: real; b: boolean); extern;
```

We will also assume that the assembly language subroutine needs four bytes of direct page work space for a pointer, which we will name PTR. Then the following equate and macro call would set things up for the assembly-language procedure:

```
Sample     start
ptr        equ      0                   work pointer

           subroutine (4:i,4:r,2:b),4
```

Notice that since the integer i is passed as a var parameter, the compiler passes a pointer to the integer, rather than the integer itself. Pointers are four bytes long. The parameter r is described to the SUBROUTINE macro as being four bytes long. When we are returning real values from a function, we pass a pointer to a ten-byte SANE extended format number. When we are passing a real, double, extended or comp number as a value parameter, however, the value is first converted to an extended number, and the ten-byte extended number is placed on the stack. Finally, boolean values require two bytes (unless they are in a packed array).

The RETURN macro has a similar protocol. If you are writing a procedure, don't code anything in the operand field. If you will be returning a value, code the number of bytes being returned (this should be two for boolean, integer and character values, and four for longint and pointers), followed by a colon, and the name of the area where the value is stored. The value must be in a direct page area or in a variable that can be accessed with absolute addressing. For example, to return a pointer called PTR, code

```
           return 4:ptr
```

Accessing Pascal Variables from Assembly Language

All variables defined at the program level are available from assembly language. If you are using the small memory model, they are all accessed using absolute addressing; in the large memory model, all variables should be accessed using absolute long addressing. Variables defined in the interface part of a unit are accessed the same way as variables in a program. Variables defined in the implementation part of a unit are also accessed the same way, but can only be used from assembly language programs appended directly onto the end of the unit. For details on how the variables are stored in memory, see Chapter 12.

It is also possible to define variables in assembly language that can be accessed from Pascal. To do that, define the variable as external in the Pascal program, using the extern directive in the variable declaration, like this:

```
var
   test: extern integer;
```

Then define the variable in assembly language. The assembly language variable must be defined globally; that is, it must be the name of a code segment, or it must be declared as global via the entry directive.

Calling Pascal Procedures and Functions from Assembly Language

Calling a procedure or function from assembly language is extremely straight forward. You simply push any required parameters onto the stack, and issue a JSL to the procedure or function you want to call. If you have called a function, two-byte values are returned to you in the accumulator; four-byte values are returned with the least significant word in the accumulator and the most significant word in the X register; and real, double, comp and extended values are returned as pointers to ten-byte SANE extended format numbers. Real values that are passed as parameters should always be pushed as ten byte extended values.

For example, to call a Pascal function that takes a pointer as input and returns an integer result, you could use the PH4 macro (supplied with ORCA/M) to push the pointer onto the stack, then call the function as follows:

```
         ph4      #parm              push the address of the parameter
         jsl      pasfunc            call the function
         sta      result             save the integer result
```

Chapter 6 – Using the Shell

Another Look at the Shell Window

The desktop development environment we have dealt with so far in this manual is very easy to use. You have probably either used or heard of some of the text based programming environments like UNIX, MS-DOS, or even the text based version of ORCA (which is included in this package). Ease of use is, of course, the biggest advantage of the desktop development environment over the text environment.

On the other hand, the text environment has several advantages over the desktop environment, too. The text environment takes less time to boot, and requires less memory. It is easy to make coding errors in Pascal that will crash the system; if you find this is true in your own programs, the shorter boot time could be significant. The shell also provides a very powerful programming tool. The shell gives you dozens of built-in commands, and even lets you add your own. You get more control over the process of compiling and linking a program with the shell, and you can even write programs, called exec files, that execute shell commands.

As it turns out, you aren't forced to choose between the desktop programming environment and the shell. You can actually use all of the features of the shell right from the desktop by simply clicking on the shell window, and typing the shell commands!

If the programs you write are generally in a single source file, you don't build libraries often, and you are not mixing Pascal with assembly language, and you are not using Apple's Rez compiler, it may not be worth your effort to learn to use the shell and the shell window. If, however, your programs fall into any of these categories, or if you would like to use the shell's impressive abilities to manage files, it would be time well spent to learn about the shell. This chapter introduces the shell, as used from the shell window in the desktop development environment. All of the topics covered, however, apply equally well to using the shell in the text environment.

Getting Into Text

While you can use the shell commands from a window on the desktop, you may want to make use of the text environment for any number of reasons. There are basically two ways to get into the text environment. The first is to set up a separate, text-based copy of ORCA/Pascal, something you can do with an installer script; see Appendix C for details if you are interested in doing this.

The other thing you can do is to set up ORCA/Pascal so you can switch between the text and desktop environment. The only change you have to make to let you switch between the two environments is to remove one line from the LOGIN file; you can find this file in the Shell folder of the ORCA/Pascal Program Disk and load it with the desktop editor. At the end of the file you will find two lines:

```
prizm
quit
```

The LOGIN file is a script file that is executed when you start up the ORCA system. The line "prizm" is a shell command that actually runs the desktop development system you have used up to this point. When you quit from PRIZM using the Quit command, you don't go right back to the Finder; instead, the shell executes the next line of the LOGIN file. In the LOGIN file that we ship with ORCA/Pascal, the next line tells the shell to quit back to the Finder. If you remove the last line, quitting from PRIZM will put you into the text shell. From there, typing quit will return you to the Finder, while typing prizm will put you back into the desktop programming environment.

After changing the LOGIN file, you will have to reboot before the shell realizes the change has been made.

How Shell Commands Work

The shell is really an interpreter, just like AppleSoft BASIC. Like AppleSoft, the shell has variables, loops, and an if statement. You can even pass variables to programs written using the shell. Unlike AppleSoft, the shell's commands are not intended for general programming. Instead, the shell has commands like catalog, which produces a detailed list of the files on a disk. The shell can manipulate files with copy (copies files or disks), move (moves files), delete (deletes files), and create (creates directories). You can see all or part of a file using type. You can also compile and link programs with a variety of commands.

You can execute shell commands from any window on the desktop. If the window you select is a shell window (that is, if the language shown in the Languages menu is Shell), you execute a command by typing the command and pressing return. In any other window, you use enter.

You can also execute groups of shell commands. To execute more than one shell command at a time, simply select the block of text containing the shell commands, then press return if you are in a shell window, or enter if you are in any other kind of window. The commands will be executed, one after the other, until all commands have executed or an error occurs.

Many shell commands write output to the screen. The "screen" is a somewhat vague term. For a variety of reasons, we usually say the output is written to "standard out." In the text environment, standard out is the text screen. When you are using the shell from the desktop environment, standard out is whatever window the shell command is issued from. Later in this chapter, you will learn how to change standard out, so that the output of a program can be sent to a disk file or printer.

Some shell commands are interactive, requiring input from the keyboard. When this happens, a cursor will appear in the window. The cursor is an inverse space. You can type in the response, and then press the return key.

File Names

When you use the desktop, you open and create files using dialogs that show you the files in a particular folder. When you are using the shell, you must type the names of files instead of using these dialogs. In all cases, the name of the file itself is the same in the shell and from the dialogs. Under the ProDOS FST, which is the one you are probably using, file names are limited to fifteen characters. Each name must start with an alphabetic character (one of the letters 'A' through 'Z'), and can contain alphabetic characters, numeric digits, or the period character in the remaining characters. You can use either uppercase or lowercase letters interchangeably.

To find a file, you need more than just the file name. Just as with the dialogs, you need to know what disk the file is on, and what folder it is in. (Folders are called directories in the text environment.) The names of disks and directories follow the same conventions as file names. The colon (or slash) character is used before the name of a disk, and between the names of disks, directories and files to separate the names from one another. Spaces are not allowed. For example, to specify the file MYFILE, located on a disk called MYDISK and in a directory called MYFOLDER, you would type

```
:mydisk:myfolder:myfile
```

It would get tiring in a hurry if you were forced to specify the name of the disk, any directories, and the file every time you wanted to refer to a disk file. Fortunately, there is a shortcut. The shell remembers the location of the directory you are currently using. If you want a file from the current directory, you only have to type the name of the file to specify the file. For example, if the current directory is :mydisk:myfolder, you only have to type **`myfile`** to get at the same file we referenced a moment ago. If the current folder is :mydisk, you would type **`myfolder:myfile`**. When you type the entire path for the file, as in :mydisk:myfolder:myfile, it is called the file's path name, or sometimes its full path name. When you use the current directory to avoid typing the full path name, as in `myfolder:myfile`, it is called a partial path name or, if no directories need to be specified at all, the file name.

You can set the current directory at any time using the prefix command. Type the name of the directory you want to become the current directory right after the name of the command. For example,

```
prefix  :mydisk:myfolder
```

sets the current prefix. Now that we are in the same directory as the file myfile (from our previous example), we can access the file by simply typing **`myfile`**. The same concept applies to directory names. Instead of using a single prefix command to set the default prefix, we could first set the prefix to the disk `:mydisk`, and then change the default prefix to the directory myfolder on that disk with the commands

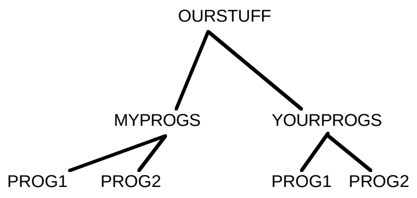
```
prefix  :mydisk
prefix  myfolder
```

In this case, the first prefix command changed the prefix to the disk `mydisk` – the leading colon tells the shell that the name is the name of a disk. The second prefix command changes the prefix to the current prefix plus the folder `myfolder`. The shell knows that the second command is changing the default prefix to a directory in the current default prefix because the name given does not start with a colon.

The current prefix is shared between the shell and the desktop. You may have noticed that when you use any of the file dialogs from the desktop, they always come up showing the folder where the last file command was executed. The desktop uses the current prefix to do this. If you use one of the file dialogs from the desktop, you can change the current prefix, and changing the current prefix from the shell will change the folder that is shown the next time you use a file dialog.

Directory Walking

Sometimes it is useful to go back a directory. The symbol .. (two periods) means go back (or up) one directory. Suppose that you have the directory structure shown below.



Assume that the current prefix is /ourstuff/myprogs. If you want to access prog1 in the yourprogs directory, you can use the partial path

```
..:yourprogs:prog1
```

to get to it. The partial path name given tells the shell to move up one directory level, from :ourstuff:myprogs to :ourstuff, and then move down the directory tree to `yourprogs:prog1`.

Device Names

GS/OS assigns a device name to each I/O device currently on line. These device names can be used as part of the path name. Let's check to see what assignments have been made. Enter the command:

```
show units
```

This command will display a table showing the device names associated with the devices on line. For an example, suppose you have a hard disk, a floppy disk, and a RAM disk installed in your computer. When you issue the show units command, you will see something like

```
Units Currently On Line:

    Number  Device              Name

    .D1     .APPLESCSI.HD01.00  :HARD.DISK
    .D4     .CONSOLE            <Character Device>
    .D6     .NULL               <Character Device>
    .D7     .PRINTER            <Character Device>
```

You can substitute a device name or a device number anywhere you would have used a volume name. Thus,

```
catalog  .d1
```

will have the same effect as

```
catalog  :hard.disk
```

Incidentally, the catalog command is a good one to know about. The catalog command lists all of the files in a directory, along with a great deal of information about each file.

Standard Prefixes

The shell provides prefixes which can be substituted for path names. We've already looked at one of these, the default prefix. There are a total of 31 of these prefixes. You can obtain a listing of the standard prefixes for your system by typing the command

```
show  prefix
```

ORCA will respond by printing a list similar to the one below.

```
System Prefix:

Number          Name

*               :ORCA.PASCAL:
@               :ORCA.PASCAL:
8               :ORCA.PASCAL:
9               :ORCA.PASCAL:
10              .CONSOLE:
11              .CONSOLE:
12              .CONSOLE:
```

```
13              :ORCA.PASCAL:LIBRARIES:
14              :ORCA.PASCAL:
15              :ORCA.PASCAL:SYSTEM:
16              :ORCA.PASCAL:LANGUAGES:
17              :ORCA.PASCAL:UTILITIES:
18              :ORCA.PASCAL:
```

The left-hand column of the listing is the prefix number. The right-hand column is a path name. The purpose of the prefix numbers is to provide you with a typing short-cut when you use path names. For example, suppose you have a program with the file name myprog located in `:ORCA.PASCAL`. You could use the path name

```
18:myprog
```

and it would have the same effect as

```
:orca.pascal:myprog
```

Notice that we have used the prefix command two ways. If you supply a prefix number followed by a path name, the prefix command changes the prefix number you give. If you type a prefix name with no prefix number, the prefix command sets the default prefix (prefix 8).

While you can modify prefix seven to suit your needs, the other prefixes have special, predefined uses. For example, if you kept your programs in a directory called `MYSTUFF`, you could rename prefix 18 to correspond to `:ORCA:MYSTUFF` using the command:

```
prefix 18 :orca:mystuff:
```

Now, when you want to access the program myprog, instead of using the path name

```
:orca:mystuff:myprog
```

you can use the path name

```
18:myprog
```

As we mentioned a moment ago, many of these prefixes have predefined, standard uses, such as defining the location of the languages prefix, or telling the linker where to look for libraries. The predefined uses are:

*	The asterisk indicates the boot prefix. The boot prefix is the name of the disk that GS/OS executed from.
@	This prefix is a special prefix used by programs that need to access user-specific information in a networked environment.

0-7	These seven prefixes are obsolete. They can only hold path names up to 64 characters. They should not be set while using ORCA/Pascal.
8	This is the default (or current) prefix. Whenever you supply a partial path name to the shell, or directly to GS/OS via a program that makes GS/OS calls, the partial path name is appended to the default prefix.
9	Prefix 9 is the program's prefix. Whenever a program is executed, prefix 9 is set to the directory where the program was found.
10	Prefix 10 is the device or file from which standard input characters are read.
11	Prefix 11 is the device or file to which standard output characters are written.
12	Prefix 12 is the device or file to which error output characters are written.
13	Prefix 13 is the library prefix. The ORCA linker searches the library prefix for libraries when unresolved references occur in a program. The Pascal compiler looks in this folder for another folder called ORCAPascalDefs to resolve uses statements.
14	Prefix 14 is the work prefix. This is the location used by various programs when an intermediate work file is created. If a RAM disk is available, this prefix should point to it.
15	Prefix 15 is the shell prefix. The command processor looks here for the LOGIN file and command table (SYSCMND) at boot time. If you use the text based editor, it also looks here for the editor, which in turn looks for its macro file (SYSEMAC), tab file (SYSTABS) and, if present, editor command table (SYSECMD). The desktop development system also uses the SYSTABS file, but does not make use of the SYSEMAC file or the SYSECMD file.
16	Prefix 16 is the languages prefix. The shell looks here for the linker, assembler, and compilers.
17	Prefix 17 is the utilities prefix. When a utility is executed, the command processor looks here for the utility. Help files are contained in the subdirectory HELP.
18-31	These prefixes do not have a predefined use.

Using Wild Cards

One of the built-in features that works with almost every command in ORCA is wild cards in file names. Wild cards let you select several files from a directory by specifying some of the letters in the file name, and a wild card which will match the other characters. Two kinds of wild cards are recognized, the = character and the ? character. Using the ? wild card character causes the system to confirm each file name before taking action, while the = wild card character simply takes action on all matching file names.

To get a firm grasp on wild cards, we will use the enable and disable commands. These commands turn the file privilege flags on and off, something that is very much like locking and unlocking files in BASIC, but with more flexibility. The privilege flags can be examined in the catalog command display. The flags are represented by characters under the access attribute. First, disable delete privileges for all files on the `:ORCA.PASCAL` directory. To do this, type

```
disable d =
```

Cataloging `:ORCA.PASCAL` should show that the D is missing from the access column of each directory entry. This means that you can no longer delete the files. Now, enable the delete privilege for the ORCA.Sys16 file. Since the ORCA.Sys16 file is the only one that starts with the character O, you can do this by typing

```
enable d O=
```

The wild card matches all of the characters after O.

What if you want to specify the last few characters instead of the first few? The wild card works equally well that way, too. To disable delete privileges for the ORCA.Sys16, you can specify the file as =Sys16. It is even possible to use more than one wild card. You can use =.= to specify all files that contain a period somewhere in the file name. Or, you could try M=.=S to get all files that start with an M, end in an S, and contain a period in between. As you can see, wild cards can be quite flexible and useful.

To return the `:ORCA.PASCAL` disk to its original state, use the command

```
ENABLE D ?
```

This time, something new happens. The system stops and prints each file name on the screen, followed by a cursor. It is waiting for a Y, N or Q. Y will enable the D flag, N will skip this file, and Q will stop, not searching the rest of the files. Give it a try!

Four minor points about wild cards should be pointed out before you move on. First, not all commands support wild cards every place that a file name is accepted. The compile, link and run commands don't allow them at all, and rename and copy commands allow them only in the first file name. Secondly, wild cards are only allowed in the file name portion, and not in the subdirectory part of a full or partial path name. For example, :=:STUFF is not a legal use of a wild card. The next point is that not all commands respect the prompting of the ? wild card. Catalog does not, and new commands added to the system by separate products may not. Finally, some commands allow wild cards, but will only work on one file. The edit command is a good example. You can use wild cards to specify the file to edit, but only the first file that matches the wild card file name is used.

Required and Optional Parameters

There are two kinds of parameters used in shell commands, required and optional. If you leave out an optional parameter, the system takes some default action. For example, if you use the catalog command without specifying a path name, the default prefix is cataloged. An example of a required parameter is the file name in the edit command: the system really needs to have a file name, since there is no system default. For all required parameters, if you leave it out, the system will prompt for it. This lets you explore commands, or use commands without needing to look them up, even if you cannot remember the exact order of all of the required parameters.

At first glance, it may seem strange to have an edit command in the shell. Its original use was to start the text editor, back in the days when the desktop development environment did not exist. You can still use it for that in the text environment, but there is also another use. If you use edit from a shell window, the file is loaded into a new window. If the file was already on the desktop, it is brought to the front. This can have several uses, especially in script files.

Redirecting Input and Output

The Apple IIGS supports two character-output devices and one character-input device. Input redirection lets you tell ORCA to take the characters from a file instead of the .CONSOLE device (which is, basically, the text screen and keyboard). When you write a character, you have a choice of two devices: standard output or standard error output. Normally, both send the characters to the screen. ORCA lets you redirect these devices separately to either a disk file or a printer.

For example, when you specify a help command, the output is printed on the screen. Using redirection, the output can be moved, or redirected, somewhere else. There are two devices that come with ORCA/Pascal that you might want to use for redirected output, or you can redirect output to any file. The first device is .PRINTER, a character device driver that comes with ORCA/Pascal that can be installed in your system folder using the Installer. Once installed, your Pascal programs can redirect output to .PRINTER to print files, or even open .PRINTER as a file from within a Pascal program to print simple text streams to your printer. The other driver is .NULL, which accepts input and does nothing; you can redirect output to .NULL if you want to execute a command, but don't want to see the output.

If you have a printer connected and turned on, and you have installed the .PRINTER driver, you can try a simple redirection:

```
help delete >.printer
```

If you do not have a printer connected, the system will hang, waiting for a response from the printer.

There are five types of redirect commands available on the command line.

<	Redirect input.
>	Redirect output.
>&	Redirect error output.
>>	Redirect output and append it to the contents of an existing file.
>>&	Append error output to an existing file.

Pipelines

Pipelines let you "pipe" the output from one process into the input for another process. The symbol for the pipeline is a vertical bar (|). For example, you might have two programs. The first program will determine the students' scores for the year. The second program will use the end-of-year scores to compute class statistics. You could use the command

```
prog1|prog2
```

instead of the series of commands

```
prog1  >data

prog2  <data
```

As another example, assume you have a program called UPPER which reads characters from the keyboard, converts them to uppercase, and writes them to the screen. Then

```
catalog  |  upper
```

would catalog your disk in uppercase.

Unlike pipelines on multitasking systems, pipelines on the Apple IIGS execute sequentially. Each program runs to completion, sending its output to a temporary file on the work prefix. The next program uses that file as its input, sending its output (if it is piped) to another temporary file. The files are called SYSPIPE0, SYSPIPE1, and so on. They are not deleted after the commands execute, so you can edit the files when debugging programs.

Writing Your Own Utilities

One of the powerful features of the shell is that you can add new commands. To do this, you simply write a normal program, then follow a few simple steps to make the shell aware of it. The program then becomes a utility. There are a variety of things that you need to know about programs that are designed to run from the shell which can help you write standard types of utilities that end up looking like they were always a part of the system. This section covers those facts, as well as stepping you through the installation of a simple utility.

Any program launcher that is capable of launching an EXE file (one kind of executable file the shell can run) is required to do some things for you. It sets up a text device for input and output, gets a user ID number for memory management calls, and if the program launcher is a shell, like ORCA or APW, it can pass the command line to you.

If you are initializing tools, you will notice that many of them ask you to reserve memory for them, usually in bank zero. When you do this, you should always use the user ID number returned by the ORCA/Pascal UserID function. This built-in function returns an integer, and requires no parameters. This function works from all environments, regardless of which program launcher executed your program. It also works for S16 files (described later). It is very important that you use this user ID number, since failure to do so can result in memory not being deleted properly when your program has finished executing.

Many program launchers, including ORCA and APW, provide an eight-character shell identifier to tell you what shell you are running under. For both ORCA and APW, the shell identifier is BYTEWRKS. You can read the shell identifier using the predefined procedure ShellID, which accepts an eight-character string as a parameter, and places the shell identifier into the string. If the program launcher does not provide a shell identifier, the string is set to eight spaces. You can use this string to see what program launcher launched you.

The last piece of information passed to you by a shell is the one most commonly used by a text based application. When you execute a program from ORCA, you type the program name, followed by some parameters. This command line, with any input and output redirection removed, is passed to you. You can read the information using the predefined procedure CommandLine, which takes a 255-character string as the parameter. The characters in the command line are placed in the string, with blanks at the end.

When you detect a run-time error in your program, you should report the error by returning a value from main, which is the error code to be returned to the shell. The error code is used by the shell to determine what steps need to be taken, if any, because of the error. For example, the shell might need to stop execution of an EXEC file. If a system error occurred, return the error number reported by the toolbox. If an internal error was detected by your program, then you should return the value -1. You should always return a value from main when you are writing shell utilities, returning 0 if there was no error.

You can find a small sample program that shows these ideas at work on your Samples disk; the path is :Samples:Text.Samples:CLine.pas. It prints the user ID number, shell identifier and command line passed to it when it executes. It then sends a zero to indicate that no error has occurred and returns. Try running the program with a variety of things typed after the command name, especially input and output redirection.

Installing a New Utility

Once you have an executable file that runs under the ORCA shell, you may want to install it as a utility. The advantages of doing so are that the program can be executed from any directory without typing a full path name, and the utility shows up in the command table. Once it is in the command table, you can use right-arrow expansion to abbreviate the command (from the text environment only), and the help command will list it.

Installing the program as a utility is really quite simple. To do so:

1. Place the program (the executable image) in the utility prefix. As shipped, this is the :ORCA.PASCAL:Utilities prefix, but you may have moved it to your hard disk, if you are using one.

2. Add the program name to the command table. The command table is in the SYSTEM folder. It is called SYSCMND. The command table is a text file, and can be changed with the editor. Simply edit it, and add the name of your program to the list of commands you see. Be sure the name of your command is the same as the file name you used for the executable file, and that the command name starts in column 1. After at least one space, type a U, which indicates that the command is a utility.

   Be sure to put the command in the correct location. The order that commands appear in the command table determines how right-arrow expansion works from the text based shell. The shell expands the first command that matches all letters typed. In general, the commands should be listed alphabetically.

   The new command will not be in the command table until you use the COMMANDS command to reread the command table, or reboot.

3. If you would like to have on-line help for the command, add a text file to the Utilities:Help folder. The name of the file must be the same as the name of the utility.

Learning More About the Shell

While this chapter has introduced the basic concepts needed to deal with the shell, we have really only scratched the surface of what the shell can do for you. After you get a little experience with shell commands and file names, you should browse through Chapter 8, which covers the shell in detail. There you will find out many more things about the shell, like how to write shell programs, and how to control the process of compiling programs more closely.

Chapter 7 – Desktop Reference

Basic Operations

The desktop development environment is a standard implementation of a desktop program, as recommended by Apple Computer. All of the basic operations that you have come to expect on the Apple IIGS and Macintosh computers are supported. Refer to the introductory manuals that came with your computer for information about the standard desktop interface.

The Cursor

The Insertion Point

The main purpose of the mouse is to position the cursor. Use the mouse to move the cursor around on the screen, and notice how the cursor changes in different regions. When it is within the confines of the text portion of the window (called the content region of the window), the cursor looks like a cross-hair. This shape allows you to use the mouse to pinpoint the location of the cursor. The selected place is called the insertion point – any typing you now do will appear before the insertion point. Notice that the insertion point is marked with a flashing vertical bar.

For example, if the line

```
Now is t|e time
```

is on the screen, you would first set the insertion point to the position shown by moving the mouse until the cursor is positioned between the 't' and 'e,' and then click the mouse. When you type a character, the text on the screen will be moved apart to make room for the new character, and the character that you typed will be placed in the space. Typing an 'h' would change the line to be

```
Now is th|e time
```

Notice how the insertion point is now between the 'h' and 'e.'

Over Strike Mode

What we have been discussing is how text is inserted into a file. The editor is normally in insert mode, but you can change this to over strike mode. When you are using the over strike mode, new characters replace the character the cursor is on, rather than moving old text over to make room for new characters. You can switch between the insert and over strike modes by using the Over Strike command in the Extras menu. When you are in the over strike mode, the insertion point will change to a line that appears under the character that will be replaced. Like the vertical bar, this line flashes.

Selecting Text

Another important use of the mouse is to select text. There are a variety of reasons to select text, including:

- Selected text can be deleted using the delete key or the Clear command. (You can retrieve the last text that you deleted by issuing the Undo command, located in the Edit menu.)
- Selected text is replaced when you type a character from the keyboard, or when you paste text from the clipboard using the Paste command.
- If any text is selected when you use the Print command, only the selected text is printed. This lets you print part of a text window without the need to copy the part you want to print to a separate window.
- If any text is selected when you use the enter key from a text window, or the return key from a shell window, the selected text is executed. Without this ability, you would be limited to executing single-line shell commands.

Selection By Dragging

Your Apple IIGS Owner's Manual described text selection by the clicking and dragging method. (That is, you click the mouse where you want to start selecting, and then drag the mouse until you have finished selecting. If you move the mouse off of the text in any direction, the page will start to scroll. This allows you to select more text than you can see in the window at any one time.) ORCA/Desktop supports this method of text selection, and also provides some short-hand ways to choose text blocks. A selection can be canceled with a single click of the mouse.

Selecting Lines

When you are typing in a program, one of the most important shortcuts is selecting a line. To select a line, start by moving the mouse to the left edge of the window. When you have moved the mouse to the left of all of the text, but while it is still on the window, you will see the cursor change to an arrow. Unlike the arrow that you see when you are selecting menu commands, this one points up and to the right. This special arrow tells you that you are in the correct place to select a line.

To actually select the line, move the mouse so it is to the left of the line you want to select, and click. The entire line appears highlighted in inverse video.

You can also select more than one line using this basic method. To select more than one line, start as you did before, by moving the mouse to the left of the first line you want to select. This time, though, hold the mouse button down and drag the mouse up or down. As you drag the mouse, all of the lines between the original line and the line you are on will be selected. As with dragging the mouse over characters, you let up on the mouse button to complete the selection.

Selecting the Entire Document

There are two ways to select all of the text in a file. The first, and simplest, is to use the Select All command, located in the Edit menu.

The second method is closely related to selecting lines. As with line selections, you start by moving the cursor to the left of the text, but keeping it in the window. The special right-arrow cursor lets you know you are in the correct place. Now, hold down the command key (the one with the on it) and click the mouse. All of the text in the document is selected. Note that it doesn't matter what line you started on.

Selecting Words

Word selection allows you to quickly isolate a single word. To do this, move the cursor so that it is on the word you want to select, and click the mouse rapidly two times. This is called double-clicking. The word that the mouse was on is selected.

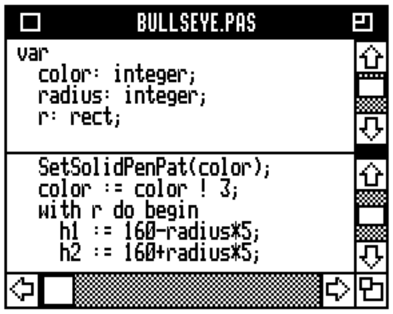
Extending a Selection

Extending a selection is a method that is generally used to select large pieces of text, although it can also be used to change the amount of text already selected. The basic idea is fairly simple. You place the cursor at one end of the text you want to select, or you use one of the existing selection methods to select some text. Now move the mouse to the point in the text where the selection is to end. (You can use scrolling or the Goto command, located in the Find menu.) Hold down the shift key, and click the mouse or continue selecting text. All of the text, from the original insertion point to the new position, is selected.

This method of selecting text is very useful when copying or deleting subroutines from a program. While you can easily drag the selection region to select the subroutine, it can take a fair amount of time to scroll the screen on a large subroutine. Instead, you can start by placing the cursor at the beginning of the subroutine, or perhaps by selecting the first line. Now move to the end using whatever method is appropriate. Holding down the shift key, select the last line in the subroutine. All lines from the first to the last are also selected, and you can easily copy or cut the subroutine from the file.

Split Screen

How many times have you been typing in a program, and wanted to refer back to an earlier subroutine or data declaration? Split screen is a feature designed to help you do that. When you split the screen, you can look at two different parts of a file at the same time.



```
var
  color: integer;
  radius: integer;
  r: rect;
```

```
  SetSolidPenPat(color);
  color := color ! 3;
  with r do begin
    h1 := 160-radius*5;
    h2 := 160+radius*5;
```

Splitting the screen is very simple. The screen splitting control is the small black box that appears just above the vertical scroll bar. Move the cursor to this box, and drag it about halfway down the page. When you release the mouse button, the screen will split.

You can edit in either half of the window. Simply use the cursor to position the insertion point, or scroll using either vertical scroll bar. The active half of the screen will change automatically.

There is one limitation on split screens. In order to show a complete scroll bar, you must have at least five lines of text in both the top and bottom half of the screen. If you try to make either part of the screen smaller, the split will be moved to give the appropriate number of lines. If the window isn't large enough to split it with five lines on both the top and bottom, the split screen control will vanish. With this restriction in mind, you can split the screen between any two lines.

Removing the split screen is just as easy as splitting it. Simply drag the split screen control to the top of the window and release it.

Entering Text

Whenever a text window is the front (active) window, and a dialog box is not active, any text you type from the keyboard will appear in the window. In insert mode, the text always appears before the insertion point. In over strike mode, the character that is underlined is replaced.

If you select some text, and then begin typing, the selected text is deleted, and the new characters appear where the selected text was located.

If the insertion point is not on the screen when you start typing, the screen will scroll to show the insertion point, and then the characters are inserted.

Special Keys

The Return Key

For any text window that is not a shell window, the return key breaks a line at the point where the return key is pressed, moving all of the text from the insertion point to the end of the line to a new line. If you are at the end of a line when you type the return key, a new, blank line is created. There are, however, many variations on this basic theme. If you are in over strike mode, the behavior of the return key changes. Instead of breaking the line or creating a new line, the return key functions simply as a cursor movement command – the insertion point is moved to the start of the next line in the file. Only if you are at the end of the file does the return key create a new line.

In block-structured languages like C and Pascal, indenting is often used to show the structure of a program. The major problem with indenting is moving the cursor to the correct spot in the line before starting to type in text. The way the return key works can be changed to make this process easier. Once changed, pressing return causes the insertion point to automatically space over, following the indentation of the line above the current line. If the current line is blank, the cursor is moved to line up with the first line above the current line that is not blank. This is called the Auto-Indent mode. To activate auto-indent mode, select Auto Indent from the Extras menu. Auto indent is turned off by selecting it a second time.

Delete Key

If you have selected any text, the delete key works exactly like the Clear command: it removes the selected text from the file. If no text is selected, the delete key deletes the character to the left of the insertion point. If the insertion point is at the start of a line, the remainder of the line is appended to the end of the line above.

Tab Key

If you are in insert mode, the tab key inserts spaces until the insertion point reaches the next tab stop. In over strike mode, the tab key simply moves the insertion point forward to the next tab stop.

The Arrow Keys

The four arrow keys can be used to move the insertion point. Using the arrow keys will deselect any previously selected text without removing it from the file.

Screen Moves

Holding down the key while typing the up-arrow key will cause the selection point to move to the top of the window. If the insertion point is already at the top of the window, the window will scroll up by one screen.

Likewise, holding down the key while typing the down-arrow key will move the selection point to the bottom of the window. Again, if you are already at the bottom of the window, the display scrolls down one screen full toward the end of the file.

Word Tabbing

You can move to the start of the next word or previous word in the file using word tabbing. A word is defined as any sequence of characters other than spaces and end-of-line markers. To move to the next word in the file, hold down the option key and type the right-arrow key. Using the left-arrow key instead of the right-arrow key will move to the beginning of the previous word.

Moving to the Start or End of a Line

You can move to the start of a line by holding down the key and typing the left-arrow key. This moves to the first column in the line, regardless of the current auto-indent mode. To move to the end of the line, hold down the key and type the right-arrow key. This moves to the column immediately after the last non-blank character in the line.

Moving Within the File

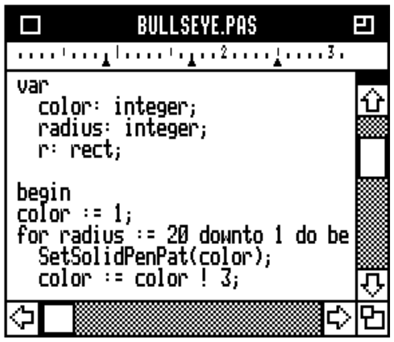
Typing one of the digit keys (1 to 9) while holding down the key will move the display to one of nine evenly spaced intervals in the file. 1 moves to the start of the file, while 9 moves to the end of the file. The other keys each move to a location one-eighth of the way through the file from the previous key.

The Ruler

You can see where the current tab stops are, and change them, by using the ruler. To make the ruler visible, use the Show Ruler command in the Extras menu. Select the same command a second time to make the ruler disappear.

With the ruler visible, your edit window will look like the one shown on the right.

The numbers, dots, and vertical bars across the top indicate the columns in the document. Every ten columns, a number appears. The twentieth column, for example, is marked with the number 2. Halfway between each numbered column is a vertical bar. The remaining columns are marked with a dot.



```
var
  color: integer;
  radius: integer;
  r: rect;

begin
color := 1;
for radius := 20 downto 1 do be
  SetSolidPenPat(color);
  color := color ! 3;
```

Under some of the columns you will see an inverted triangle pointing at the column marker. This inverted triangle is a tab stop. When you use the tab key, it moves the insertion point to the next tab stop, inserting a tab character if you are in insert mode or past the end of the line in overstrike mode.

To remove an existing tab stop, move the cursor so that the arrow points at the tab stop, and click. To create a tab stop where none exists, move the cursor to the column on the ruler where you want a tab stop, and click.

Moving a tab stop, then, is a two-step process. First, remove the old tab stop, and then place a new tab stop in the proper column. Of course, the order of these steps can be changed.

Default tab stops

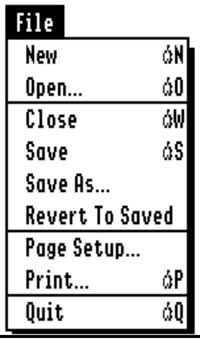
All of the ORCA language development environments are multi-lingual; the same environment can be used with more than one language. Tab stops that are reasonable for assembly language, however, may not be the best choice for Pascal. The same is true for virtually any pair of languages you might pick.

As a result, each language has a different default tab line. When you open a new window, the tab line is set to the default tab stops for the language assigned to the new window. If you change the language stamp, a dialog will appear that gives you the choice of changing to the new language's default tabs or sticking with the ones that are already in use.

The default tab line is changed by making changes in the SYSTABS file. This is described in detail later in this chapter. For now, the important point is that changing the tab stops with the ruler does not change the default tabs. The next time you load the file from disk, the original tab stops will again be used.

The File Menu

The File menu is used to open files, save files to disk that have been created or changed with the editor, quit the program, and for various disk-based housekeeping functions.



New

The New command opens a new window. Until it is saved for the first time, the window will be called "Untitled X," where X is a number. The first new window opened will be assigned a number of 1, and subsequent windows will increment the value. You would use the New command to create new programs.

Open

The Open command is used to open a text file that already exists on a disk. After choosing the file from the Open command's file list, it will be opened and placed in a new window. Like all windows newly created on the desktop, this window will be as large as the screen.

Close

The Close command closes the front window. The front window is the window that is currently highlighted. If the file has changed since the last time it was saved, a dialog box appears before the window is closed. The dialog box gives you a chance to save the changes to the file, or to cancel the close operation.

This menu item will be dimmed if there are no windows open on the desktop.

Save

If the front window was loaded from disk, or if it has already been saved at least one time, then ORCA knows which disk file is associated with the window. In that case, this command causes the contents of the window to be written to the disk. After the write operation is complete, the desktop returns to its original state – the file is still on the desktop, and all characteristics of the file have been preserved.

If you use the Save command on an untitled window, it will function as though you had selected the Save As command. The Save As command is discussed below.

Save As

The Save As command is used to write the contents of a window to a file that is different from the original text file, or to save a new, untitled window to a file for the first time. The Save As... dialog is the standard file save dialog, described in the manuals that come with your computer.

Revert To Saved

The contents of the window are replaced by a copy of the file read from disk. The cursor is moved to the first character of the first line of the file, but all other options (such as over strike or auto indent) remain the same.

This menu will be dimmed if there have been no changes to the file.

Page Setup

The Page Setup command is used when you are ready to print the contents of one of your open windows. The actual dialog depends on the printer driver you have selected from the chooser. For detailed information about the Page Setup dialog, see the documentation that comes with your computer.

Print

The Print command sends the contents of the front window to your printer. You can select only a portion of your document to be printed, or, if no text has been selected when you issue the Print command, then the entire file will be printed.

The Print command brings up a standard dialog to control the printing process. This dialog is documented in the manuals that came with your computer.

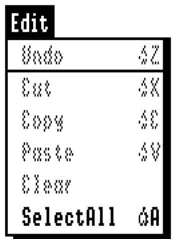
Quit

All windows on the desktop are closed. If any of the files have changed since the last time they were saved, you are presented with a dialog box that gives you a chance to save the file or cancel the Quit command. If you cancel the Quit command, all windows that have already been closed stay closed. Once all windows are closed, the program returns control to the text programming environment. From there, you can use the shell's quit command to return to the program launcher that you used to start ORCA/Pascal.

The Edit Menu

The Edit menu provides the standard editing capabilities common to virtually all desktop programs. You can select all of the text in the document; cut, copy or clear selected text; paste text from the current scrap; or undo changes to the file.

Undo



The Undo command changes the file back to the state it was in before the last command that changed the file was executed. For example, if you use the delete key to delete several characters of text, then use the Undo command, the deleted characters will reappear in the file.

If you have enough memory to hold all of the changes, repeated use of the Undo command will eventually return the file to the same condition it was in when it was originally loaded from disk. If memory starts to run short, all but the most recent changes may be lost. In general, you should not depend on being able to undo more than one command.

Cut

The selected text is removed from the file and placed in the clipboard. You can paste this text anywhere in a window with the Paste command, described below. The clipboard holds only one block of text at a time. The next Cut or Copy command will cause the contents of the clipboard to be replaced by the new selection.

This menu item is disabled if no text has been selected.

Copy

The selected text is copied to the clipboard, replacing the previous contents of the clipboard. The file being edited is not affected.

This menu item is disabled if no text has been selected.

Paste

The contents of the clipboard are copied into the file at the current insertion point. If any text was selected when the Paste command was issued, the selected text is cleared before the Paste is performed.

Clear

The selected text is removed from the file.

Assuming that some text has been selected, this command is equivalent to using the delete key.

This menu item is disabled if no text has been selected.

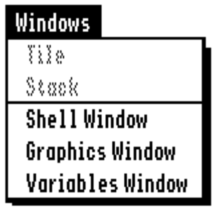
Select All

All of the text in the file is selected.

You can also select all of the text in the file by moving the mouse to the left of the text, holding down the command (⌘) key, and clicking the mouse.

The Windows Menu

The Windows menu gives you control over how the windows are displayed, and helps you find windows on a cluttered desktop. The Tile and Stack commands sort the files on the desktop into two different pictorial formats. The names of all windows currently open are also shown in this menu. The front window's name is marked with a check. You can bring any window to front by selecting its name from the window list.



Tile

The Tile command changes all of the windows on the desktop to the same size, then places them so that none overlap. The name comes from the fact that the windows are placed next to one another, much as tiles are laid down on a floor.

Tiling the windows is a quick way to organize your desktop. Once the windows are tiled, it is fairly easy to find a particular window. On the other hand, if you have a lot windows open, they generally become too small to be useful. That is when the zoom box, located at the top right of the window's title bar, becomes handy. When you click this box, the window grows to take up the entire desktop. Once you have finished with the window, and would like to select another, click the zoom box again. The window returns to its original size and location, and you can see all of the tiled windows again.

If there are nine windows on the desktop, the Tile command will create three rows; each row will have three windows. If more than nine windows are on the desktop, the extras are laid on top of the first nine.

Stack

The Stack command stacks the windows on the desktop. Each window is moved a little to the right of the window it covers, and it is also moved far enough down so that the window's name can be read.

If there are more than seven windows open, the extra windows are stacked on top of the first seven windows.

Shell Window

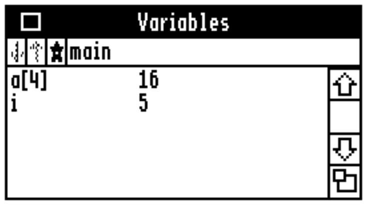
The Shell Window command opens the shell window. The shell window is basically an untitled window with a few special characteristics. The special characteristics are: the shell window has the name Shell, rather than a name that starts with Untitled; it shows up in the top right corner of the screen; and the shell window always starts with a language stamp of Shell.

This window will be opened automatically before any EXE program, including the compiler, is executed.

Graphics Window

The Graphics Window command brings up a special window where the output from graphics programs can be written without leaving the programming environment. Whenever you write a graphics program, use this command to open the window before running your program. If you forget to open the graphics window, the program will still run, but the graphics output will be lost.

Variables Window



```
a[4]          16
i             5
```

The Variables Window command brings up another special window. You can enter the names of variables from your program, and the variable and its current value will show up in the window, updating as you step through your program. When debugging a program, you would normally select the Variables command, and then select one of the debugging commands such as Step, Trace, or Go. You cannot enter a variable name until the program begins execution, since variables are undefined until run-time. Also, the variable names that you type into the Variables window can only be entered when the program is executing in the subroutine for which these variables are defined.

The Variables window above is typical. Under the window's title bar are an up-arrow, a down-arrow, star, and the name of the currently executing subroutine or main program. Beneath the arrows is a list of variable names and their current values. Along the right side of the window is a scroll bar, used to scroll through the variables list.

The arrows next to the current subroutine's name can be used to move through the local variables in the various subroutines; they cannot be selected unless your program is executing at a point where a function call has been detected by the debugger. For example, once you enter a subroutine from the program body, the window display changes to show the variables in the subroutine. The up-arrow darkens, indicating that you may click on it to change the display to that of the main program. If you select the up-arrow, you will see the variables display that you created in the program block, and the down-arrow can be selected so that you may return to the subroutine's variables display.

The star button is a short-cut that displays all of the simple variables available from the current subroutine. Simple variables are any variable that does not need to be dereferenced with an array subscript, pointer operator, or field name.

You can enter variable names by clicking anywhere in the content region of the Variables window. After clicking, a line-edit box appears under the subroutine-name box. You can enter the name of one variable in the box, using any of the line-edit tools to type the name. Press the return key after entering the name, and the variable's current value will be immediately displayed to the right of the name. If you decide later that you need to edit or delete the variable name, then click on the name and use any line-edit tools you need to accomplish the task.

Only the names of specific values may be entered into the Variables window; you cannot view the contents of structures or entire arrays. It is possible to see the value of any array declared as an array of characters, however. In that case, the debugger expects a null-terminated string.

When you display a pointer, you will see its value printed in hexadecimal format. You can also look at the value if the object pointed to by the pointer. To do this, place a ^ character after the pointer's name.

The contents of individual array elements can be seen in the Variables window, provided that the array elements are scalar types. You must enter all of the indices associated with an array element (i.e. an element in a four-dimensional array requires four indices). An array element is specified by first entering the name of the array, and then the indices enclosed in either parentheses or square brackets. While the desktop will recognize both parentheses and square brackets, the opening and ending punctuation must match. (i.e. use '(' with ')' and '[' with ']').

You can look at any field within a record or object by typing the record or object name, a dot, and the name of the field.

If a pointer points to a record or object, you can look at a field in the record or object by typing the name of the pointer, then either ^. or ->, and finally the field name.

These dereference operators can be used in combination. For example, it is possible to look at an element of an array that is in a record pointed at by a pointer with a sequence like this one:

```
ptr^.arr[4];
```

The names entered into the variables window are case insensitive – leNGTh and LEngth would be the same variable name, for example.

Any spaces you type are left in the string for display purposes, but are otherwise ignored, even if they appear in an identifier.

The debugger can display variables which are stored internally in any of the following formats:

- 1-, 2-, and 4-byte integers
- 4-, 8-, and 10-byte reals
- Pascal style strings and null terminated strings
- booleans
- characters (Only the first byte of the character is examined. Nonprinting characters are output as blanks.)
- pointers (These print as hexadecimal values.)

Table 7.1: Variable formats

The variables window is updated after each command is executed by a Step or Trace command. It is also updated when a break-point is encountered. The variables window is not updated if the Go command is used, or during the execution of a Step Through or Go To Next Return command.

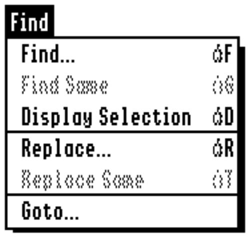
List of Window Names

As you open windows, their names appear after the Stack command in the Windows menu. When you pull down the Windows menu, you can see a list of all of the windows on the desktop, by name. The window that you are using when you pull the menu down is checked.

If you would like to use a different window, you can select it from the windows list. The window you select is placed on top of all of the other windows on the desktop, and becomes the active window.

There is only room for eleven window names in the Windows menu. If there are more than eleven windows on the desktop, the extra names will not be displayed in the windows list.

The Find Menu

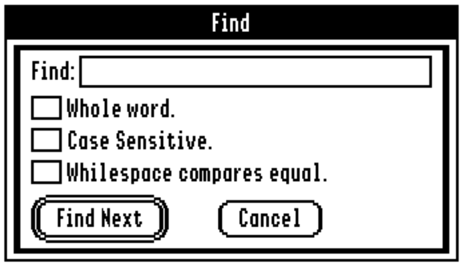


The Find menu helps you locate strings in a window, replace occurrences of a string with another string, find the cursor, or move to a particular line by line number.

The Find menu does not appear on the menu bar unless there is a file open on the desktop.

Find

The Find command is used to find a sequence of characters in the current window.



When you select the Find command, a dialog like the one above appears on your screen. The Find dialog is a modal dialog that stays in place until one of the buttons is selected. To find a string of text, enter the text in the line-edit box next to Find: and click on the Find Next button (or press the return key). The window display will change as necessary to show the first occurrence of the string after the current insertion point, and the string will be selected.

You can continue searching for the same string by continuing to click on the Find Same command. If the end of the program is reached, the search starts over at the beginning of the file. The only time the search will fail is if there are no occurrences of the search string in the entire document.

There are three options that affect the way searching is conducted. These appear as check boxes in the Find window. The first is Whole Word. When selected, this option will only find strings that are preceded by a non-alphanumeric character or occur at the beginning of a line, and that end in a non-alphanumeric character or the end-of-line marker. For example, searching for the word "int" with Whole Word enabled would find a match in both of these lines:

```
int i;
{i is int}
```

The characters "int" in this line, though, would not be found:

```
{print this line}
```

The Case Sensitive option makes the search case sensitive. That is, searching for INT would not find the word int.

In many situations, especially when programming in assembly language, you want to find two words separated by some spaces. For example, if you want to find the line

```
           lda   #2
```

you really don't care if there are two or three spaces between the two words – you just want to find all of the places where the accumulator is loaded with the constant 2. In this case, you would want to use the White Space Compares Equal option. When selected, all runs of spaces and tabs are treated as if they were a single space.

Find Same

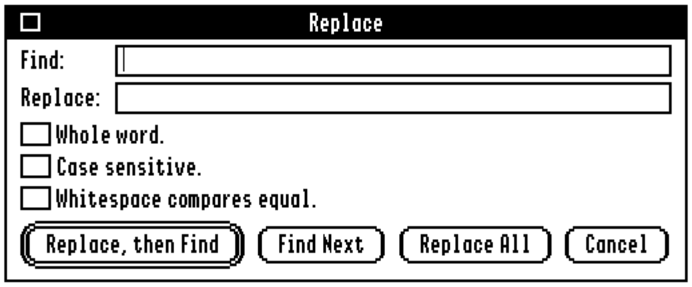
If you have already entered a search string using the Find command, you can search for the next occurrence of the same string using the Find Same command. This allows you to avoid using the Find dialog, and enables searching by simply using the G keyboard equivalent.

Display Selection

If the insertion point (or selected text) is visible on the screen, this command does nothing. If you have used the scroll bars to move the display so that the insertion point does not appear on the screen, the Display Selection command moves the display so that you can see the insertion point.

Replace

The Replace command brings up a dialog like the one shown below. All of the buttons, check boxes, and line-edit box from the Find command are present, and are used the same way. In addition, there are two new buttons and one new text box:



To use the Replace command, you enter a search string exactly as you would with the Find command. In fact, if you have already used the Find command, the search string you had entered will appear in the Replace window, and the state of the check boxes will also be the same. Enter a replacement string in the Replace: box. You can move to this box with the cursor or with the tab key. Set the options you want with the check boxes.

If you would like to replace all occurrences of the search string with the replacement string, you can click on the Replace All button. To examine the target strings before deciding whether to replace them, use the Find Next button. If you decide that you do want to change the current target string, then click the Replace, then Find button. This button will also cause the search to continue after replacement.

After you have found and/or replaced a string, you might want to continue editing your document. To return to your document window, you must either close the Replace window or bring your program window to front. To use the Replace command again, you can make it the active window by clicking anywhere on the Replace window (assuming this window is visible), or you can reissue the Replace command.

Replace Same

Once you have entered Find and Replace strings with the Replace command, you can use the Replace Same command to replace a single occurrence of the target string. The Replace Same command is equivalent to the Replace then Find button in the Replace dialog. This avoids use of the Replace window, and allows you to replace strings with a single keystroke (this command's keyboard equivalent is T). In conjunction with the G keyboard equivalent for Find, you can quickly scan through a program, replacing any occurrences of a string.

Goto

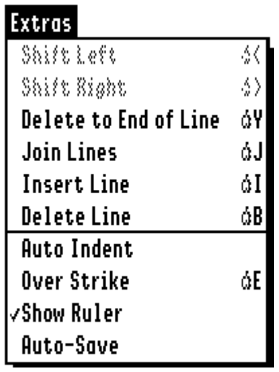
The Goto command lets you move to any line in the open file by specifying a line number. The line number is entered as a decimal value in the Goto window's line-edit box. Clicking on the Goto button causes the desired line to appear at the top of the window, with the insertion point changed to the beginning of this line. The Cancel button just causes the Goto window to vanish.

Goto is very useful when you are looking through a list of errors written to the shell window by a compiler or assembler. Most of these listings show line numbers along with the line where the error occurred.

The Extras Menu

The Extras menu has several editing commands not found in the standard Edit menu. These commands allow you to shift blocks of text from a block-structured program to the left and right, perform several complex editing operations (like deleting all characters from the cursor to the end of the line), and set several editing options.

The Extras menu does not appear on the menu bar unless there is a file open on the desktop.



Shift Left

When you are programming in a block-structured language, like C or Pascal, indentation is usually used to show the structure of the program at a glance. If the structure changes, you may want to change the indentation of large blocks of text. The Shift Left command, along with the Shift Right command described below, can help.

The Shift Left command is only available if you have selected some text. Regardless of whether you selected entire lines or not, the Shift Left command works on whole lines, not on characters. It scans all of the lines that have at least one character selected, and deletes one space from the beginning of the line. The effect is to move a block of selected text left by one column. Only spaces are deleted – if a line has already been shifted as far to the left as possible, it is left untouched.

Shift Right

Like the Shift Left command, described above, Shift Right is used to move blocks of text. The Shift Right command is only available if you have selected some text. All of the lines in the file that have at least one character selected are moved to the right by inserting a space before the first character in the line.

If any of the lines are 255 characters long before this command is used, the last character on each of the long lines will be lost.

Delete to End of Line

If any text is selected, it is cleared from the file. Next, all of the characters from the insertion point to the end of the line are deleted.

Join Lines

If any text is selected, it is cleared from the file. The line after the one the cursor is on is then removed from the file, and appended to the end of the line containing the cursor. The insertion point is placed between the two joined lines.

If the combined line has more than 255 characters, all of the characters past the 255th character are lost.

Insert Line

If any text has been selected, it is cleared. Next, a new, blank line is inserted in the file beneath the line containing the current insertion point.

Delete Line

If any text has been selected, it is cleared. Next, the line containing the current insertion point is deleted from the file.

Auto Indent

When you are writing programs in a block-structured language, like C or Pascal, indentation is often used to show program structure. The Auto Indent option can help you indent your programs.

If the auto indent mode has not been selected, pressing the return key causes the insertion point to move to the beginning of the next line. If you are in over strike mode, hitting the return key will not affect the current line; the insertion point simply moves to the start of the next line in the file. If you are in insert mode, the current line is split, and the cursor moves to the start of a new line. This function is provided for assembly language and other line-oriented languages.

When you select the Auto Indent option, the return key works a little differently. Instead of moving to the first column of a line, it spaces over to match the current indentation. If over strike has also been selected, the cursor moves to the first non-blank character in the next line. If the line is blank, the cursor is aligned with the first non-blank character in the line above.

With the over strike option turned off, but with auto indent turned on, the cursor still moves so that it is under the first non-blank character in the line above. If a line has been split, blanks are inserted to move the insertion point to the proper column.

Over Strike

The editor is capable of operating in one of two modes, insert or over strike. Insert mode is the most common mode for desktop programs, so it is the default mode. In insert mode, all characters typed are inserted into the window by first inserting a new space to the left of the insertion point, then placing the new character in the space.

Text-based editors generally use over strike mode. In over strike mode, any character typed replaces the character that the cursor is on.

You can tell which mode you are in by pulling down the Extras menu. If the over strike option has a check mark next to it, you are in the over strike mode. If there is no check mark, you are in insert mode. You can also tell which mode you are in by looking at the insertion point. If the insertion point marker is a flashing vertical bar, you are in the insert mode. If it is a flashing horizontal line, you are in over strike mode.

Show Ruler

When you select the Show Ruler command, a ruler appears in an information bar at the top of the front window. The ruler has markings which show the column numbers. Below these, any tab stops appear as inverted triangles. Selecting Show Ruler a second time will remove the ruler display.

The description of the ruler, earlier in this chapter, gives more details on how to use the ruler once it is visible.

Auto Save

The Auto Save option is a safety measure. If you execute a program, and the program crashes, you cannot return to the desktop to save your files. Any changes that have been made to the files since the last time they were saved to the disk are lost.

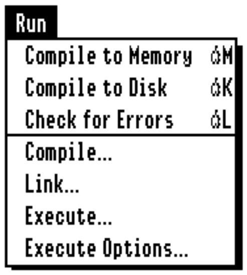
The Auto Save command can prevent this kind of catastrophe. Before executing any program, any file on the desktop that has been changed is saved to disk. This takes time – with floppy disks, the time can be considerable. For that reason, this feature is an option. Whether you select it or not should depend on how often you save your files, and how likely you think it is that your program will crash.

Keep in mind that what we mean by a crash is a catastrophic failure, where you actually end up in the monitor, or where you have to reset the computer. Normal run-time errors in compiled programs are trapped. These present you with an error message, but do not endanger any files on the desktop.

One other note of caution. Saving your files to a RAM disk provides very little protection from a nasty crash. Often, a crash is due to a program writing to memory that it has not reserved. This kind of bug is very common in programs that use the toolbox or that make use of Pascal's new and dispose procedures. It can also happen if you are using arrays and index past the end of the array. If a program is doing this, your RAM disk is no safer than files on the desktop. If you want to be sure that your files will not be lost, save them to a floppy disk or hard disk.

The Run Menu

The Run menu contains the commands that allow you to compile a program. There are a variety of ways to compile a program, reflecting options suited to different sizes of programs and differing personal taste.



Compile to Memory

The Compile To Memory command compiles, links and executes the program in the front window. Object modules are not saved to disk, but the executable file is written to disk. This command will probably be the one you will use most to compile your programs – it gives the fastest turn-around time since writing the object modules to disk is avoided.

You should not use this command if your program is split across multiple source files, and you need the object modules to combine with other object files to form the final executable file. (This is called separate compilation.) You should also not use this option if your program is made up of more than one language. For example, if you use the append directive to append an assembly language file to the end of a Pascal program, do not use this command to compile the program. In either of these cases, use Compile to Disk.

There are some compilers that do not support Compile to Memory. In these cases, you must use the Compile to Disk command, or you will get linker errors. ORCA/Pascal supports Compile to Memory.

Whenever you compile a program, information about the compilation is written to a special window called the shell window. You can create this window yourself, by selecting New from the File menu and then giving it a language stamp of shell. (See the description of the Languages menu below for more information about the language stamp.) If you have not created a shell window, the desktop will do so automatically when you compile a program for the first time.

Compile to Disk

This command compiles, links, and executes your program. Unlike Compile to Memory, the program's object files are written to disk. With that exception, it works just like the Compile To Memory command.

ORCA creates object files as a result of compiling or assembling source files; it creates executable files as the output from linking object files. The number of object files created is typically two, while there is one executable module. The first object file contains some compiler initialization code; ORCA attaches the suffix *.root* to the name it uses for this module. The second object file contains the rest of the generated intermediate code; ORCA attaches the suffix *.a* to its name. If any other object files are created, the next successive alphabetic character is appended to the file name (i.e. .b, .c, ... , .z). Multiple object modules could be created by performing some series of partial and/or separate compilations of various source files.

If your source file contains a keep directive, ORCA will use the keep name in creating the object and executable files associated with compiling your program. For example, if your keep name is OUT, then the object files will be named OUT.ROOT and OUT.A.

For programs which do not use a keep directive, ORCA uses default names for the object and executable files created as a result of compiling and linking your program; the names are derived from the name of your source file. If your source file's name contains a suffix (i.e. a period within the name, followed by one or more characters), then the system calls the first object file *sourcefile*.root, where *sourcefile* is the name of your source file, with the suffix stripped. The second object file is named *sourcefile*.a. The executable file is named *sourcefile*. If your source file's name does not contain a suffix, then ORCA appends the four-character suffix *.obj* to the output files. The first object file will have *.root* appended to the *.obj*, and the second will have *.a* appended to the *.obj*. For example, if your source file was named FILE1, then the object files would be named FILE1.OBJ.ROOT and FILE1.OBJ.A, while the executable file would be called FILE1.OBJ.

A word of caution: using the ProDOS FST, GS/OS restricts file names to 15 characters. If you will be using the default names assigned by the desktop, you need to ensure that your source file's name is not too long when the suffixes are attached to form the object and executable files' names.

Programmers typically assign suffixes to their file names to remind them of the file's language type. We recommend the following suffixes:

Language	Suffix
Pascal	.PAS
assembly	.ASM
BASIC	.BAS
C	.CC

We strongly recommend that you not use single-character suffixes, since these can interfere with partial compiles and multi-lingual compiles.

Check for Errors

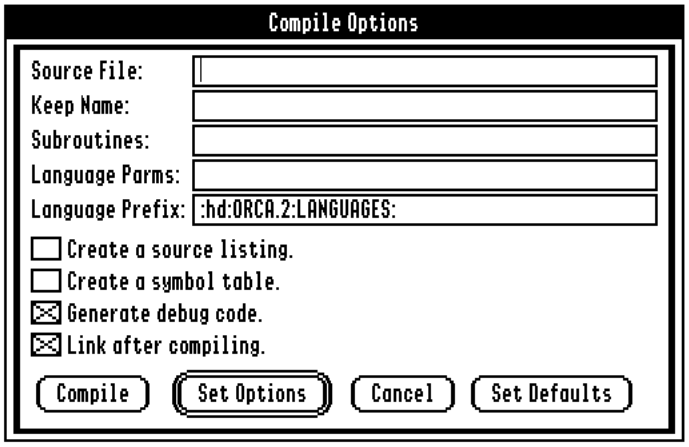
The Check For Errors command compiles your program, but does not save the result of the compile. This allows the compiler to scan your program quickly for errors. Most compilers can scan for errors about twice as fast as they can compile a program. Once all errors have been removed, you can use one of the compile commands to compile the program.

If you use a keep directive in your program, this command will compile your program instead of just scanning for errors. To make effective use of this command, be sure to remove any keep directives. Note that removal of keep directives allow you to use the automatic naming for object and executable files discussed above.

Compile

The purpose of this command is to set the default options for compilation, or to compile a program without linking. Note that the options you choose affect all compile commands selected to compile the front window.

Below is a picture of the dialog box brought up by the Compile command.


Compile Options

Source File:
Keep Name:
Subroutines:
Language Parms:
Language Prefix: :hd:ORCA.2:LANGUAGES:

☐ Create a source listing.
☐ Create a symbol table.
☒ Generate debug code.
☒ Link after compiling.

The rectangular boxes next to the first five items in the Compile window are line-edit boxes. In the Source File: box, you can enter the name of the source program that you want to compile. A complete or partial path name may be entered here.

The Keep Name: box is where you enter the name of the object module produced by compiling the source file; again, this can be either a full or partial path name. Any name supplied here takes precedence over KEEP names supplied in your source file, or over the default naming of object files described earlier in this section. Make sure the KEEP name is different from the source file name to prevent linker errors when the linker tries to overwrite the source file with the object module.

The Subroutines: box is used for partial compilation. Under ORCA, once you have compiled a complete program, you can individually compile selected subroutines. This can be very useful when you have a long program made up of several subroutines. If you find you have made a mistake in only a few of the subroutines, then you are not forced to recompile the entire program to correct these few mistakes. To perform a partial compile, enter the names of the subroutines needing to be recompiled, separated by a space. Not all compilers support partial compilation. Please refer to Chapter 8 for more information about partial compilation.

The Language Parms: box is used to tell the system about any special parameters your compiler needs. ORCA/Pascal does not use these fields. If you are using another compiler, your compiler reference manual will tell you if you need these options.

The Language Prefix: box is used to tell the system that you have installed your compilers in some directory other than the default Languages prefix. The default prefix is the subdirectory named Languages contained in the directory where you installed your desktop system. If you are using the full ORCA shell or more than one compiler, setting up a special directory to hold your compilers, assemblers, and linker is a good idea. You should enter either a full or partial path name here.

The next four boxes are check boxes. To select any or all of the options, move the cursor over the box and click once with the mouse. To deselect an option, click on the box a second time.

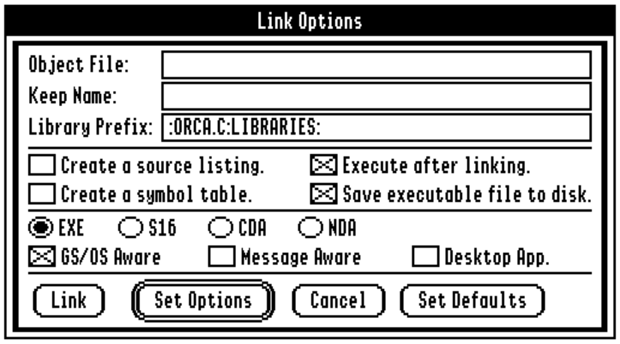
Checking the Create a source listing box causes the compiler to produce a listing of your source file as it compiles your program, and checking the Create a symbol table box causes the compiler to produce a symbol table. A symbol table is a summary of the all of the functions and variables detected in the program. ORCA/Pascal does not produce a symbol table. Generate debug code calls for the compiler to produce special code that will be used by the desktop in running the source-level debugger. The debug box should be checked while you are in the process of debugging your program, and then deactivated after your program is working properly so that the code produced by the compiler is more compact. Link after compiling causes the desktop to invoke the linker after successful compilation of your program.

The four buttons in the bottom of the Compile window cause the desktop to take action based on the button chosen. Clicking the Compile button starts the compilation of your source file. Clicking the Set Options button causes the desktop to record information about future compilations based on the choices you have made in this window. Cancel returns you to where you were before selecting the Compile command; no system action is taken. The Set Defaults button causes the desktop to record the information you have given in this dialog. Then, whenever you launch the desktop, the compilation options specified here will be automatically applied to the program being compiled. See "Setting up Defaults," later in this chapter, for further information about setting system defaults.

Link

The purpose of the Link command is to set default options to be used when linking the front window, or to manually link object modules.

The linker can be regarded as an advanced feature. You do not need to understand the function of a linker to effectively use the desktop, since the compile commands are set up to automatically call the linker.


Link Options

Object File:

Keep Name:

Library Prefix: :ORCA.C:LIBRARIES:

☐ Create a source listing.
☒ Execute after linking.
☐ Create a symbol table.
☒ Save executable file to disk.

[✓ EXE] S16 CDA NDA

☒ GS/OS Aware
☐ Message Aware
☐ Desktop App.

The line-edit box following Object File: is where you enter the base name of the object files you wish to be linked. The object file's name should not include any system-added file name extensions. For example, if you had compiled a program named BULLSEYE.PAS, using a keep name of BULLSEYE, then the system would have created object modules named BULLSEYE.ROOT and BULLSEYE.A. To link these two object modules, you would enter BULLSEYE as the name of the Object File. Default object file names are discussed above with the Compile command.

The Object File box can also be used to perform separate compilation. The first object file name you enter should contain the main program; the other names can be specified in any order. Enter only the base names of the object files, as explained in the preceding paragraph. The linker will automatically load all of the object modules produced from compiling a single source file. See Chapter 8 for more information about separate compilation.

The line-edit box following Keep Name: is where you enter the name of the executable file that the system will create upon successful linking of the object modules. It is customary, but not required, to use the same name as that given in the Object File box; the system knows which files are object modules and which are executable images because the object module names always contain system-added extensions. Using the bull's eye example above, then, we would enter BULLSEYE for the object file and BULLSEYE for the Keep Name. The object modules would be called BULLSEYE.ROOT and BULLSEYE.A, while the executable file would be named BULLSEYE.

The Library Prefix: box is used to tell the system that you have installed the libraries you and your compilers use in some directory other than the default library prefix. The default prefix is the subdirectory named LIBRARIES contained in the directory where you installed your desktop system. You must enter a full path name here.

As with the Compile window, the next four boxes are check boxes. The first box gives you the option of producing a listing of the link. The second box is used to specify whether a symbol table is to be generated during linking. The third box lets you specify whether execution of the program should immediately occur after successful linking of the object modules. The fourth box tells the system whether or not to save the executable image to disk. This last option is for future expansion; currently, the linker saves the file to disk if there is a keep name, and does not save the file if there is no keep name.

The radio buttons below the check boxes allow you to set the file type of the executable image. Different file types are used depending upon the function of the program. If you want to execute the program without leaving the development environment, use a file type of EXE. You must use EXE to use the debugger, shell window, or graphics window.

If you wish to create a stand-alone program that can be launched from the Finder, change the file type to S16, turn debugging off, and compile your program. S16 programs can be executed by the development environment, but the desktop shuts down before executing your program. S16 programs can also be executed from the Finder; EXE programs cannot.

Classic desk accessories have a file type of CDA, while new desk accessories have a file type of NDA. You can execute a new desk accessory from the desktop as if it were an EXE program, but you must still set the file type to NDA. Once the desk accessory is debugged, copy the executable image to the DESK.ACCS subdirectory of the SYSTEM directory. Remember to turn debugging off before the final compilation! After the desk accessory has been installed into the SYSTEM/DESK.ACCS directory, you can access it from the Apple menu of any desktop program.

Classic desk accessories cannot be debugged directly from the desktop. To debug a classic desk accessory, compile it as an EXE program with a main program that calls the initialization and action functions. Once debugged, remove the main program from the source code, turn off debugging, change the file type to CDA, and then recompile your program. You can then copy the finished executable program to the SYSTEM:DESK.ACCS directory, where it can be accessed by using the three-key command sequence -control-esc.

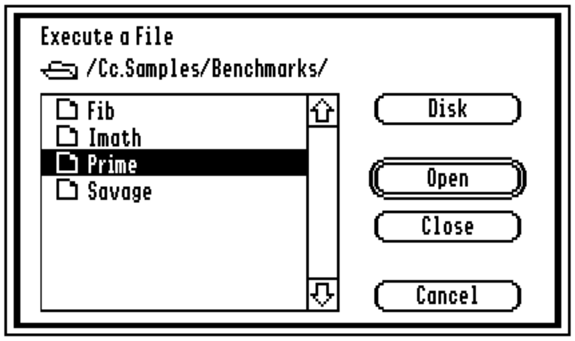
The three check boxes below the radio buttons are used to set bits in the auxiliary file type; these are used by various program launchers to decide how to execute your program. The complete description for these options is in Apple's File Type Notes for file type $B3 (S16) or $B5 (EXE). Briefly, "GS/OS Aware" tells the program launcher that your program is a modern one that knows about the longer prefixes, and will use prefix 8 for the default prefix. The ORCA/Pascal libraries assume you are using the new prefixes, so this option should be checked. "Message Aware" tells the Finder that your program uses messages passed by the message center. This would be true of most desktop programs. "Desktop App." tells the Finder that the program is a desktop application. In this case, the Finder shuts down the tools in a special way so the text screen doesn't flash on the screen as your program starts.

Clicking the Link button causes the system to begin linking the object modules named in the Object File: box. Selecting Set Options causes the information entered in the dialog to replace the previous linker defaults. The Cancel button closes the dialog without saving the changes. The

Set Defaults button causes the desktop to record the information you have given in this dialog. Then, whenever you launch the desktop, the linker options specified here will be automatically applied to the program being linked. See "Setting up Defaults," later in this chapter, for further information about setting system defaults.

Execute

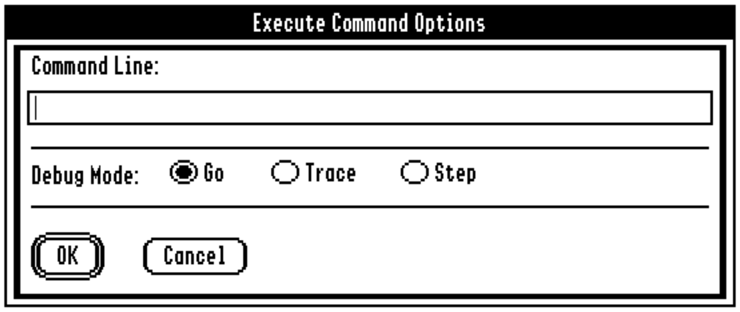
The Execute command allows you to run an executable program. The program's file type must be EXE, S16, or SYS. The dialog box brought up by this command is shown below:



To execute a file, simply open it.

Execute Options...

The Execute Options command allows you to set certain characteristics that effect the Execute command and programs with debug code that are executed from the shell window.

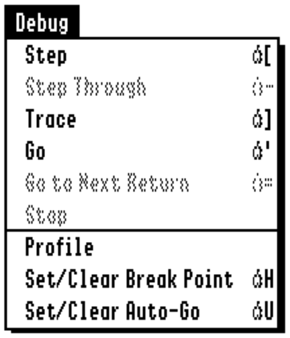


The command line is passed to the program as if it were typed from the shell window. Be sure and include the name of the program, since the program will expect to find the name. Do not use I/O redirection, piping, or multiple commands on one line.

The "Debug Mode" radio buttons tell the debugger how to execute the program. The three starting modes start the program at full speed ("Go"), in trace mode ("Trace") or in single-step mode ("Step"). If the program was compiled with debug code off, the setting of these buttons is ignored.

The Debug Menu

The Debug menu contains commands that allow you to operate the source-level debugger. All of the source-level debug options require the compiler to generate special debug code. Many compilers support the +d flag to generate this debug code. If they do not, these debugging options cannot be used. Chapter 4 has information about debugging desktop programs, as well as a tutorial on the debugger itself.



Step

When you select Step, Trace, or Go, the first thing that happens is the system checks to see if the program in the front window has been compiled. If not, the front window is compiled to memory, and then executed in the selected debug mode. If the program has already been compiled, the disk copy of the program is loaded and executed.

When you select the Step command, your program starts executing, but stops when it gets to the first executable line. A small arrow appears in the source window, pointing to the line that will be executed next. At this point, you can use any of the debugging commands.

Repeated use of the Step command steps through your program, one line at a time. As this happens, the arrow pointing to the current line will be updated. Using this method, you can actually watch your program execute, quickly locating problem spots.

If you are using a Variables window, all variable values in the window are updated after each Step command.

Step Through

If you encounter a function call while you are stepping through a program using the Step command, you will step right into the function, executing its commands one by one until it returns to the subroutine which called it. Many times, you do not want to step through each line of the subroutine. Instead, you would rather concentrate on one function, assuming that the subroutines called work correctly.

The Step Through command helps you do this. It works exactly like the Step command until you come to a line with a function call. On those lines, the function is executed at full speed. Execution of the stepped-through subroutine will be terminated if a run-time error is detected, a break point is encountered, or the Stop command is selected from the Debug menu.

Trace

When you use the Trace command, the program starts stepping automatically. The Variables window still gets updated after each line is executed, and you can still watch the flow of the program as the arrow moves through each line that is executed. At any time, you can use the Step command to stop the trace. That does not stop the execution of the program; it only pauses, waiting for the next debug command. Step, Step Through, Trace, Go, Go to Next Return, and Stop can all be used.

If you want to pause during a trace, move the cursor to the right side of the menu bar and hold the mouse button down. Program execution will cease until you let up on the mouse button. As soon as you release the mouse button, the trace will resume.

Go

When you select the Go command, your program starts executing at full speed. It will continue executing until it is finished, a break point is reached, or the Stop command is issued.

Go to Next Return

This command is used to allow a subroutine to run to completion. If you have been stepping or tracing through a subroutine, and you get to a point where you do not need to watch the remainder of the subroutine execute, simply use the Go to Next Return command. The program will execute at full speed until the end of the subroutine. You will end up in step mode, with the debugging arrow pointing to the line in the source window which comes after the line which called the subroutine.

Stop

The Stop command terminates execution of the program. Any program that was compiled with debugging turned on can be stopped this way, whether or not it was started using the debug commands.

Profile

The Profile command helps you find the functions where your program is spending the most time. It returns the following three statistics about the execution of your program: the number of times each subroutine and main program was called; the number of heartbeats that occurred during each subroutine and main program; the percent of heartbeats for each subroutine and main program as a function of the total number of heartbeats generated during the entire execution.

The Profiler is a routine which is installed into the heart-beat interrupt handler of the computer. It maintains a stack of pointers to profiling information. Upon entry to a new subroutine, the subroutine's name is added to the stack, and profiling counters are incremented. When entering a subroutine which is already included in the stack, the pointer to the subroutine's information is accessed and the appropriate counters are incremented.

The information returned by Profile can be quite accurate for some programs, but be somewhat misleading for others. The Profiler works by counting heartbeats. A heartbeat occurs 60 times each second. Each time a heartbeat occurs, the heartbeat counter for the current subroutine is incremented. If the subroutines in your program are very short, they may not take enough CPU time for a heartbeat to occur. If the program runs for a long time, the impact of this problem is reduced. Counting heartbeats is, after all, a statistical process. The larger your sample, the better the results will be.

Another potential problem area is disabling interrupts. Heartbeats are interrupts – disabling interrupts stops the process of counting heartbeats. The most common culprit is GS/OS, which disables interrupts while reading and writing to the disk.

To obtain the best results from the Profiler, then, use it on a long execution. Be suspicious of statistics for programs that have very short, fast subroutines, or that perform lots of disk I/O.

Set/Clear Break Points

Break points are used when you want to execute up to some predetermined place in your program, then pause. For example, if you know that the first 500 lines of your program are working correctly, but you need to step through the 20 lines after that, it would take a great deal of time to get to the suspected bug using the Step or Trace commands. You can, however, set a break point. You would start by setting a break point at line 500, then execute the program using one of the Compile commands. When your program reached line 500, execution would stop, and the arrow marker would point to line 500. You could then use the debug commands to examine the area of interest.

There is no limit to the number of break points that can be placed in a compiled program.

To set a break point in a compiled program, start by selecting the line or lines in the source window where a break point is to occur. With the lines selected, apply the Set/Clear Break Point command. A purple X will appear to the left of the line, indicating that the line has a break point.

To remove an existing break point, select the line and use the Set/Clear Break Point command again. The X that indicates a break point will vanish.

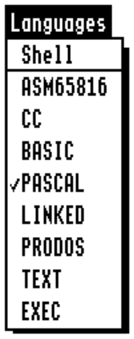
Set/Clear Auto-Go

There may be places in your program that you always want the Step and Trace commands to skip. That is where the Set/Clear Auto-Go command is used. Any lines that have been set for auto-go will execute at full speed, even if you are using the Step and Trace commands.

To mark lines for auto-go, select the lines and then invoke this command. A green spot will appear to the left of the selected lines. To clear auto-go, select the lines and apply the command again.

A line cannot be marked for both auto-go and as a break point. If you select a line for auto-go, any existing break point is removed. Similarly, marking a line for a break point will remove its auto-go status.

The Languages Menu



The Languages menu shows all of the languages installed on your system. It changes when you install or delete a programming language. You can use this menu to find out what language is associated with a particular file, or to change the language.

Under ORCA, all source and data files are associated with a language. The system uses the underlying language stamp to call the appropriate compiler or assembler when you issue a compile command for a source file.

Shell

Shell is a special entry, and so is set off from the other names in the Languages menu. The desktop maintains a window called the shell window, whose corresponding language is the Shell. You can create a window yourself, by first selecting the New command located in the File menu, and then selecting Shell from the Languages menu. If you do not create a Shell window, the desktop will create one for you the first time that you compile a program.

The desktop uses the Shell window to display information about what it is doing. For example, when you compile a program, the results of compilation are shown in the Shell window.

You can also use the Shell window to communicate with the ORCA shell. You can enter any available shell commands, and then press return. The shell will execute the commands and then return to the desktop, displaying any text output in the shell window, as well as using the shell window for prompts and to echo text responses. See Chapter 8 for a detailed description of the shell. Chapter 6 has a brief introduction to the shell, describing in more detail how to use the shell from the desktop development environment.

Installed Languages

Below the name Shell in the Languages menu is a list of the names of the compilers and assemblers that are currently installed in your desktop system, as well as some names used for other ASCII file types. Each text window in the desktop will have a language stamp associated with it. You can pull down the Languages menu to see what language stamp the front window has, or you can select a different language for the front window by selecting the appropriate language from this menu. The language associated with the front window will be checked.

There is always one language which is the current language; it is the same as the language of the front window. When you change the language stamp of the front window, you also change the current system language. New windows are stamped with the current system language.

The languages ProDOS, Text, and Exec are special. A file whose language stamp is ProDOS means that the file contains only ASCII text. Data files read by a program are typically stamped as ProDOS. The language Text is reserved for use by text editors. The language Exec is given to shell script files. See Chapter 8 for more information about Exec files.

The SYSTABS File

The SYSTABS file is located in the SYSTEM prefix of the program disk. It contains the default settings for tab stops, auto-indent mode, and cursor mode. It is an ASCII text file that can be opened under the desktop and edited to change the default settings.

Each language recognized by ORCA is assigned a language number. The SYSTABS file has three kinds of lines associated with each language:

1. The language number.
2. The default settings for the different editing modes.
3. The default tab and end-of-line-mark settings.

The first line of each set of lines in the SYSTABS file specifies the language that the next two lines apply to. ORCA languages can have numbers from 0 to 32767 (decimal). The language number must start in the first column; leading zeros are permitted and are not significant, but leading spaces are not allowed.

The second line of each set of lines in the SYSTABS filesets the defaults for various editor modes, as follows:

1. If the first column contains a zero, pressing return causes the insertion point to be placed in column one in the next line. If column one (in the SYSTABS file) contains a one, then pressing return aligns the insertion point with the first nonspace character in the next line. If the line is blank, then the insertion point is aligned with the first nonspace character in the line above.
2. The second column is used by the text-based editor to indicate the selection mode. It is not used by the desktop editor. It can be either a zero or one.
3. The next character indicates the wrap mode. It is not used by the desktop editor.

4. The fourth character is used to set the default cursor mode. A zero will cause the editor to start out in over strike mode. A one will cause it to start in insert mode.
5. The fifth and sixth characters are used by the text based editor.

The third line of each set of lines in the SYSTABS file sets default tab stops. There are 255 zeros and ones, representing the length of lines under the desktop. The ones indicate the positions of the tab stops. A two in any column sets the end of the line. The column containing the two then replaces column 255 as the default right margin when the editor is set to that language.

For example, the following lines define the defaults for ORCA/Pascal. Note that only the first few characters of the tab line are shown; the tab line actually contains 255 characters.

```
5
100110
0000000010000000001000000000100000000010000000010
```

If no defaults are specified for a language (that is, there are no lines in the SYSTABS file for that language), then the editor assumes the following defaults:

- return sends the cursor to column one.
- The editor starts in insert mode.
- There is a tab stop every eighth column.
- The end of the line is at column 255.

Setting up Defaults

You can tailor your environment on the desktop by setting various options, and saving them. Then, whenever you run the desktop, your defaults will be automatically loaded, and your desktop will look the same from session to session.

ORCA always saves information about your environment before it executes an S16 program, to ensure that everything will be as it was after execution. This allows the environment to be purged while your program executes, then have everything return to its original state when your program finishes. Automatic rebuilding of your environment saves you time, since you do not have to reopen various files and windows, size them correctly, etc. It also allows you to quickly remember what you were doing before you left the desktop.

ORCA records the following information about your current desktop, in a file named PRIZM.TEMP, located in the same prefix as PRIZM:

- The path name of the file displayed in the front window, and that window's size and location on the screen.
- The setting of the Auto-save flag.
- Where prefixes 8, 13, and 16 are located.
- The settings of the compile flags for source listing, symbol table, generation of debug code, and link after compile.

- The settings of the link flags for source listing, symbol table, saving of the executable file, and file type of the executable file.
- The setting of the Profile flag.
- The current language.

Permanent default information is stored in the file named PRIZM.CONFIG, located in the same folder as PRIZM. The same information listed above is saved. To set these defaults, use the Save Defaults button from the Compile or Link dialogs. To return to the system defaults, simply delete the PRIZM.CONFIG file.

Chapter 8 - The Command Processor

This chapter will cover the operation of the ORCA Command Processor. A command processor is an interface between you and the operating system of a computer. You enter a command on the command line. The command processor will interpret your command and take some specific action corresponding to your command. The command processor for ORCA is very powerful. The features available to you and discussed in this chapter are:

- The line editor.
- Command types.
- Standard prefixes and file names.
- EXEC files.
- Input and output redirection.
- Pipelines.
- Command table.
- Command reference.

The Line Editor

When commands are issued to the shell, they are typed onto the command line using the line editor. The line editor allows you to:

- Expand command names.
- Make corrections.
- Recall the twenty most recently issued commands.
- Enter multiple commands.
- Use wildcards in file names.

Command Name Expansion

It is not necessary to enter the full command name on the command line. Type in the first few letters of a command (don't use RETURN) and press the RIGHT-ARROW key. It will compare each of the commands in the command table with the letters typed so far. The first command found that matches all of the characters typed is expanded in the command line. For example, if you typed:

```
CORIGHT-ARROW
```

ORCA would match this with the command COMMANDS, and would complete the command like this:

```
COMMANDS
```

Editing A Command On The Command Line

The available line-editing commands available are listed in the table below:

command	command name and effect
LEFT-ARROW	**cursor left** - The cursor will move to the left on the command line.
RIGHT-ARROW	**cursor right** - The cursor will move to the right. If the cursor is at the end of a sequence of characters which begin the first command on the line, the shell will try to expand the command.
LEFT-ARROW	**word left** - The cursor will move to the start of the previous word. If the cursor is already on the first character of a word, it moves to the first character of the previous word.
RIGHT-ARROW	**word right** - The cursor will move to the end of the current word. If the cursor is already on the last character in a word, it moves to the last character in the next word.
UP-ARROW or DOWN-ARROW	**edit command** - The up and down arrows are used to scroll through the 20 most recently executed commands. These commands can be executed again, or edited and executed.
> or .	**end of line** - The cursor will move to the right-hand end of the command line.
< or ,	**start of line** - The cursor will move to the left-hand end of the command line.
DELETE	**delete character left** - Deletes the character to the left of the cursor, moving the cursor left.
F or CTRLF	**delete character right** - Deletes the character that the cursor is covering, moving characters from the right to fill in the vacated character position.
Y or CTRLY	**delete to end of line** - Deletes characters from the cursor to the end of the line.
E or CTRLE	**toggle insert mode** - Allows characters to be inserted into the command line.
Z or CTRLZ	**undo** - Resets the command line to the starting string. If you are typing in a new command, this erases all characters. If you

	are editing an old command, this resets the command line to the original command string.
ESCX or CLEAR or CTRLX	**clear command line** - Removes all characters from the command line.
RETURN or ENTER	**execute command** - Issue a command to the shell, and append the command to the list of the most recent twenty commands.

Table 8.1 Line-Editing Commands

The shell normally starts in over strike mode; see the description of the {Insert} shell variable to change this default.

The shell's command line editor prints a # character as a prompt before it accepts input. See the description of the {Prompt} shell variable for a way to change this default.

Multiple Commands

Several commands can be entered on one line using a semicolon to separate the individual commands. For example,

```
RENAME WHITE BLACK;EDIT BLACK
```

would first change the name of the file WHITE to BLACK, and then invoke the editor to edit the file named BLACK. If any error occurs, commands that have not been executed yet are canceled. In the example above, if there was an error renaming the file WHITE, the shell would not try to edit the file BLACK.

Scrolling Through Commands

Using the UP-ARROW and DOWN-ARROW keys, it is possible to scroll through the twenty most recent commands. You can then modify a previous command using the line-editing features described above and execute the edited command.

Command Types

Commands in ORCA can be subdivided into three major groups: built-in commands, utilities, and language names. All are entered from the keyboard the same way.

Built-in Commands

Built-in commands can be executed as soon as the command is typed and the RETURN key is hit, since the code needed to execute the command is contained in the command processor itself. Apple DOS and Apple ProDOS are examples of operating systems that have only built-in commands.

Utilities

ORCA supports commands that are not built into the command processor. An example of this type of command is CRUNCH, which is a separate program under ORCA. The programs to perform these commands are contained on a special directory known as the *utilities* directory. The command processor must first load the program that will perform the required function, so the *utilities* directory must be on line when the command is entered. The command will also take longer to execute, since the operating system must load the utility program. Most utilities can be restarted, which means that they are left in memory after they have been used the first time. If the memory has not been reused for some other purpose, the next time the command is used, there is no delay while the file is loaded from disk.

The utilities themselves must all reside in the same subdirectory so that the command processor can locate them. The name of the utility is the same as the name of the command used to execute it; the utility itself can be any file that can be executed from the shell, including script files. Utilities are responsible for parsing all of the input line which appears after the command itself, except for input and output redirection. The command line is passed to a utility the same way it is passed to any other program.

Language Names

The last type of command is the language name. All source files are stamped with a language, which can be seen when the file is cataloged under ORCA. There is always a single system language active at any time when using ORCA.

The system language will change for either of two reasons. The first is if a file is edited, in which case the system language is changed to match the language of the edited file. The second is if the name of a language is entered as a command.

Table 8.2 shows a partial list of the languages and language numbers that are currently assigned. CATALOG and HELP will automatically recognize a language if it is properly included in the command table. ProDOS has a special status: it is not truly a language, but indicates to the editor that the file should be saved as a standard GS/OS TXT file. Language numbers are used internally by the system, and are generally only important when adding languages to ORCA. They are assigned by Apple Computer, Inc.

language	number
ProDOS	0
TEXT	1
ASM6502	2
ASM65816	3
ORCA/Pascal	5
EXEC	6
ORCA/C	8

Table 8.2 A Partial list of the Languages and Language Numbers

You can see the list of languages currently installed in your system using the SHOW LANGUAGES command. While all of the languages from the above table are listed, the compiler needed to compile C programs and the assembler needed to assemble ASM65816 programs are sold separately.

Program Names

Anything which cannot be found in the command table is treated as a path name, and the system tries to find a file that matches the path name. If an executable file is found, that file is loaded and executed. If a source file with a language name of EXEC is encountered, it is treated as a file of commands, and each command is executed, in turn. Note that S16 files can be executed directly from ORCA. ProDOS 8 SYSTEM files can also be executed, provided ProDOS 8 (contained in the file P8) is installed in the system directory of your boot disk.

Standard Prefixes

When you specify a file on the Apple IIGS, as when indicating which file to edit or utility to execute, you must specify the file name as discussed in the section “File Names” in this chapter. GS/OS provides 32 prefix numbers that can be used in the place of prefixes in path names. This section describes the ORCA default prefix assignments for these GS/OS prefixes.

ORCA uses six of the GS/OS prefixes (8 and 13 through 17) to determine where to search for certain files. When you start ORCA, these prefixes are set to the default values shown in the table below. You can change any of the GS/OS prefixes with the shell PREFIX command, as described in this chapter.

GS/OS also makes use of some of these numbered prefixes, as does the Standard File Manager from the Apple IIGS toolbox. Prefixes 8 through 12 are used for special purposes by GS/OS or Standard File. Prefix 8 is used by GS/OS and Standard File to indicate the default prefix; that's the same reason ORCA uses prefix 8. Prefix 9 is set by any program launcher (including GS/OS, ORCA, and Apple's Finder) to the directory containing the executable file. Prefixes 10, 11 and 12 are the path names for standard input, standard output, and error output, respectively. Use of these prefixes is covered in more detail later in this chapter.

Prefix Number	Use	Default
@	User's folder	Boot prefix
*	Boot prefix	Boot prefix
8	Current prefix	Boot prefix
9	Application	Prefix of ORCA.Sys16
10	Standard Input	.CONSOLE
11	Standard Output	.CONSOLE
12	Error Output	.CONSOLE
13	ORCA library	9:LIBRARIES:
14	ORCA work	9:
15	ORCA shell	9:SHELL:
16	ORCA language	9:LANGUAGES:
17	ORCA utility	9:UTILITIES:

Table 8.3 Standard Prefixes

The prefix numbers can be used in path names. For example, to edit the system tab file, you could type either of the following commands:

```
EDIT  :ORCA:SHELL:SYSTABS
EDIT  15:SYSTABS
```

Each time you restart your Apple IIGS, GS/OS retains the volume name of the boot disk. You can use an asterisk (*) in a path name to specify the boot prefix. You cannot change the volume name assigned to the boot prefix except by rebooting the system.

The @ prefix is useful when you are running ORCA from a network. If you are using ORCA from a hard disk or from floppy disks, prefix @ is set just like prefix 9, defaulting to the prefix when you have installed ORCA.Sys16. If you are using ORCA from a network, though, prefix @ is set to your network work folder.

The current prefix (also called the default prefix) is the one that is assumed when you use a partial path name. If you are using ORCA on a self-booting 3.5 inch disk, for example, prefix 8 and prefix 9 are both normally :ORCA:. If you boot your Apple IIGS from a 3.5-inch :ORCA disk, but run the ORCA.Sys16 file in the ORCA: subdirectory on a hard disk named HARDISK, prefix 8 would still be :ORCA: but prefix 9 would be :HARDISK:ORCA:.

The following paragraphs describe ORCA's use of the standard prefixes.

ORCA looks in the current prefix (prefix 8) when you use a partial path name for a file.

The linker searches the files in the ORCA library prefix (prefix 13) to resolve any references not found in the program being linked. ORCA comes with a library file that supports the standard Pascal library; you can also create your own library files.

The resource compiler and the DeRez utility both look for a folder called RInclude in the library prefix when they process partial path names in include and append statements. The path searched is 13:RInclude. See the description of the resource compiler for details.

When the compiler encounters a uses statement, and a LibPrefix directive has not been used, it searches first in 13:ORCAPascalDefs.

The work prefix (prefix 14) is used by some ORCA programs for temporary files. For example, when you pipeline two or more programs so that the output of one program becomes the input to the next, ORCA creates temporary files in the work prefix for the intermediate results (pipelines are described in the section “Pipelines” in this chapter). Commands that use the work prefix operate faster if you set the work prefix to a RAM disk, since I/O is faster to and from memory than to and from a disk. If you have enough memory in your system to do so, use the Apple IIGS control panel to set up a RAM disk (be sure to leave at least 1.25M for the system), then use the PREFIX command to change the work prefix. To change prefix 14 to a RAM disk named :RAM5, for example, use the following command:

```
PREFIX 14 :RAM5
```

You won't want to do this every time you boot. You can put this command in the LOGIN file, which you will find in the shell prefix. The LOGIN file contains commands that are executed every time you start the ORCA shell.

ORCA looks in the ORCA shell prefix (prefix 15) for the following files:

EDITOR
SYSTABS
SYSEMAC
SYSCMND
LOGIN

As we mentioned a moment ago, the LOGIN file is an EXEC file that is executed automatically at load time, if it is present. The LOGIN file allows automatic execution of commands that should be executed each time ORCA is booted.

ORCA looks in the language prefix (prefix 16) for the ORCA linker, the ORCA/Pascal compiler, and any other assemblers, compilers, and text formatters that you have installed.

ORCA looks in the utility prefix (prefix 17) for all of the ORCA utility programs except for the editor, assembler, and compilers. Prefix 17 includes the programs that execute utility commands, such as CRUNCH and MAKELIB. The utility prefix also contains the HELP: subdirectory, which contains the text files used by the HELP command. Command types are described in the section “Command Types and the Command Table” in this chapter.

Prefixes 0 to 7

The original Apple IIGS operating system, ProDOS 16, had a total of eight numbered prefixes that worked a lot like the 32 numbered prefixes in GS/OS. In fact, the original eight prefixes, numbered 0 to 7, are still in GS/OS, and are now used to support old programs that may not be able to handle the longer path names supported by GS/OS.

When the programmers at Apple wrote GS/OS, one of the main limitations from ProDOS that they wanted to get rid of was the limit of 64 characters in a path name. GS/OS has a theoretical limit of 32K characters for the length of a path name, and in practice supports path names up to 8K characters. This presented a problem: existing programs would not be able to work with the longer path names, since they only expected 64 characters to be returned by calls that returned a path name. Apple solved this problem by creating two classes of programs: GS/OS aware programs, and older programs. When a program launcher, like Apple's Finder or the ORCA shell, launches a GS/OS aware program, prefixes 0 to 7 are cleared (if they had anything in them to start with). The program launcher expects the program to use prefixes 8 and above. When an old program is executed, prefixes are mapped as follows:

GS/OS prefix	old ProDOS prefix
8	0
9	1
13	2
14	3
15	4
16	5
17	6
18	7

In each case, the new, GS/OS prefix is copied into the older ProDOS prefix. If any of the GS/OS prefixes are too long to fit in the older, 64 character prefixes, the program launcher refuses to run the old application, returning an error instead. Assuming the old application is executed successfully, when it returns, the old ProDOS prefixes are copied into their corresponding GS/OS prefixes, and the ProDOS prefixes are again cleared.

The ORCA shell fully supports this new prefix numbering scheme. When you are working in the ORCA shell, and use a prefix numbered 0 to 7, the ORCA shell automatically maps the prefix into the correct GS/OS prefix. The shell checks for the GS/OS aware flag before running any application, and maps the prefixes if the application needs the older prefix numbers.

File Names

File name designation in ORCA follows standard GS/OS conventions. There are some special symbols used in conjunction with file names:

symbol	meaning
.Dx	This indicates a device name formed by concatenating a device number and the characters '.D'. Use the command: **SHOW UNITS**

	to display current assignment of device numbers. Since device numbers can change dynamically with some kinds of devices (e.g. CD ROM drives) it is a good idea to check device numbers before using them.
.name	This indicates a device name. As with device numbers, the "show units" command can be used to display a current list of device names. The two most common device names that you will use are .CONSOLE and .PRINTER, although each device connected to your computer has a device name. .CONSOLE is the keyboard and display screen, while .PRINTER is a device added to GS/OS by the Byte Works to make it easy for text programs to use the printer.
x	Prefix number. One of the 32 numbered prefixes supported by GS/OS. See the previous section for a description of their use. You may use a prefix number in place of a volume name.
..	When this is placed at the start of a path name, it indicates that the reference is back (or up) one directory level.
:	This symbol, when inserted in a path name, refers to a directory. You can also use /, so long as you do not mix : characters and / characters in the same path name.

ORCA allows the use of a physical device number in full path names. For example, if the SHOW UNITS command indicates that the drive with the disk named :ORCA is .D1, the following file names are equivalent.

```
:ORCA:MONITOR     .D1:MONITOR
```

Here are some examples of legal path names:

```
:ORCA:SYSTEM:SYSTABS
..:SYSTEM
15:SYSCMND
.D1
.D3:LANGUAGES:ASM65816
14:
```

Wildcards

Wildcards may be used on any command that requires a file name. Two forms of the wildcard are allowed, the = character and the ? character. Both can substitute for any number of characters. The difference is that use of the ? wildcard will result in prompting, while the = character will not. Wildcards cannot be used in the subdirectory portion of a path name. For example,

```
    DELETE  MY=
```

would delete all files that begin with MY.
The command,

```
    DELETE  MY?
```

would delete files that begin with MY after you responded yes to the prompt for each file. The wildcards can be used anywhere in the file name.
There are limitations on the use of wildcards. Some commands don't accept wildcards in the second file name. These commands are:

COPY
MOVE
RENAME

There are some commands that only work on one file. As a result, they will only use the first matching file name. These commands are:

ASML
CMPL
CMPLG
COMPILE

Types of Text Files

GS/OS defines and uses ASCII format files with a TXT designator. ORCA fully supports this file type with its system editor, but requires a language stamp for files that will be assembled or compiled, since the assembler or compiler is selected automatically by the system. As a result, a new ASCII format file is supported by ORCA. This file is physically identical to TXT files; only the file header in the directory has been changed. The first byte of the AUX field in the file header is now used to hold the language number, and the file type is $B0, which is listed as SRC when cataloged from ORCA.
One of the language names supported by ORCA SRC files is TEXT. TEXT files are used as inputs to a text formatter. In addition, PRODOS can be used as if it were an ORCA language name, resulting in a GS/OS TXT file. TXT files are also sent to the formatter if an ASSEMBLE, COMPILE, or TYPE command is issued.

EXEC Files

You can execute one or more ORCA shell commands from a command file. To create a command file, set the system language to EXEC and open a new file with the editor. Any of the commands described in this chapter can be included in an EXEC file. The commands are executed in sequence, as if you had typed them from the keyboard. To execute an EXEC file, type the full path name or partial path name (including the file name) of the EXEC file and press RETURN.

There is one major advantage to using an EXEC file over typing in a command from the command line. The command line editor used by the shell restricts your input to 255 characters. With EXEC files, you can enter individual command lines that are up to 64K characters in length. Since it probably isn't practical or useful to type individual command lines that are quite a bit wider than what you can see on your computer screen, you can also use continuation lines. In any EXEC file, if the shell finds a line that ends with a backslash (\) character (possibly followed by spaces or tabs), the line is concatenated with the line that follows, and the two lines are treated as a single line. The command is treated exactly as if the backslash character and the end of line character were replaced by spaces. For example, the command

```
link file1 file2 file3 keep=myfile
```

could be typed into an EXEC file as

```
link             \
   file1         \
   file2         \
   file3         \
   keep=myfile
```

The two versions of the command would do exactly the same thing.

If you execute an interactive utility, such as the ORCA Editor, from an EXEC file, the utility operates normally, accepting input from the keyboard. If the utility name was not the last command in the EXEC file, then you are returned to the EXEC file when you quit the utility.

EXEC files are programmable; that is, ORCA includes several commands designed to be used within EXEC files that permit conditional execution and branching. You can also pass parameters into EXEC files by including them on the command line. These features are described in the following sections.

EXEC files can call other EXEC files. The level to which EXEC files can be nested and the number of variables that can be defined at each level depend on the available memory.

You can put more than one command on a single line of an EXEC file; to do so, separate the commands with semicolons (;).

Passing Parameters Into EXEC Files

When you execute an EXEC file, you can include the values of as many parameters as you wish by listing them after the path name of the EXEC file on the command line. Separate the parameters with spaces or tab characters; to specify a parameter value that has embedded spaces or tabs, enclose the value in quotes. Quote marks embedded in a parameter string must be doubled.

For example, suppose you want to execute an EXEC file named FARM, and you want to pass the following parameters to the file:

cow
chicken
one egg
tom's cat

In this case, you would enter the following command on the command line:

```
FARM cow chicken "one egg" "tom's cat"
```

Parameters are assigned to variables inside the EXEC file as described in the next section.

Programming EXEC Files

In addition to being able to execute any of the shell commands discussed in the command descriptions section of this chapter, EXEC files can use several special commands that permit conditional execution and branching. This section discusses the use of variables in EXEC files, the operators used to form boolean (logical) expressions, and the EXEC command language.

Variables

Any alphanumeric string up to 255 characters long can be used as a variable name in an EXEC file. (If you use more than 255 characters, only the first 255 are significant.) All variable values and parameters are ASCII strings of 65535 or fewer characters. Variable names are not case sensitive, but the values assigned to the variables *are* case sensitive. To define values for variables, you can pass them into the EXEC file as parameters, or include them in a FOR command or a SET command as described in the section “EXEC File Command Descriptions.” To assign a null value to a variable (a string of zero length), use the UNSET command. Variable names are always enclosed in curly brackets ({}), except when being defined in the SET, UNSET and FOR commands.

Variables can be defined within an EXEC file, or on the shell command line before an EXEC file is executed, by using the SET command. Variables included in an EXPORT command on the shell command line can be used within any EXEC file called from the command line. Variables included in an EXPORT command within an EXEC file are valid in any EXEC files called by that file; they can be redefined locally, however. Variables redefined within an EXEC file revert to their original values when that EXEC file is terminated, except if the EXEC file was run using the EXECUTE command.

The following variable names are reserved. Several of these variables may have number values; keep in mind that these values are literal ASCII strings. A null value (a string of zero length) is considered undefined. Use the UNSET command to set a variable to a null value. Several of the predefined variables are used for special purposes within the shell.

{0}	The name of the EXEC file being executed.

{1}, {2}, ...	Parameters from the command line. Parameters are numbered sequentially in the sequence in which they are entered.
{#}	The number of parameters passed.
{AuxType}	Provides automatic auxiliary file type specification. The variable contains a single value, specified as a hex or decimal integer. The AuxType string sets the auxiliary file type for the executable file produced by the linker. Any value from 0 to 65535 ($FFFF) can be used.
{CaseSensitive}	If you set this variable to any non-null value, then string comparisons are case sensitive. The default value is null.
{Command}	The name of the last command executed, exactly as entered, excluding any command parameters. For example, if the command was :ORCA:MYPROG, then {Command} equals :ORCA:MYPROG; if the command was EXECUTE :ORCA:MYEXEC, then {Command} equals EXECUTE. The {Parameters} variable is set to the value of the entire parameters list.
{Echo}	If you set this variable to a non-null value, then commands within the EXEC file are printed to the screen before being executed. The default value for Echo is null (undefined).
{Exit}	If you set this variable to any non-null value, and if any command or nested EXEC file returns a non-zero error status, then execution of the EXEC file is terminated. The default value for {Exit} is non-null (it is the ASCII string true). Use the UNSET command to set {Exit} to a null value (that is, to delete its definition).
{Insert}	When you are using the shell's line editor, you start off in over strike mode. If the {Insert} shell variable is set to any value, the shell's line editor defaults to over strike mode.
{KeepName}	Provides an automatic output file name for compilers and assemblers, avoiding the KEEP parameter on the command line and the KEEP directive in the language. If there is no keep name specified on the command line, and there is a non-null {KeepName} variable, the shell will build a keep name using this variable. This keep name will be applied to all object modules produced by an assembler or compiler. On the ASML, ASMLG and RUN commands, if no {LinkName} variable is used, the output name from the assemble or compile will also determine the name for the executable file. See {LinkName} for a way to override this.

	There are two special characters used in this variable that affect the automatic naming: % and $. Using the % will cause the shell to substitute the source file name. Using $ expands to the file name with the last extension removed (the last period (.) and trailing characters).
{KeepType}	Provides automatic file type specification. The variable contains a single value, specified as a hex or decimal integer, or a three-letter GS/OS file type. The KeepType string sets the file type for the executable file produced by the linker. Legal file types are $B3 to $BF. Legal file descriptors are: EXE, S16, RTL, STR, NDA, LDA, TOL, etc.
{Libraries}	When the linker finishes linking all of the files you specify explicitly, it checks to see if there are any unresolved references in your program. If so, it searches various libraries to try and resolve the references. If this variable is not set, the linker will search all of the files in prefix 13 that have a file type of LIB. If this variable is set, the linker searches all of the files listed by this shell variable, and does not search the standard libraries folder.
{LinkName}	Provides an automatic output name for the executable file created by the link editor. The % and $ metacharacters described for {KeepName} work with this variable, too. When an ASML, ASMLG or RUN command is used, this variable determines the name of the executable file, while {KeepName} specifies the object file name. This variable is also used to set the default file name for the LINK command.
{Parameters}	The parameters of the last command executed, exactly as entered, excluding the command name. For example, if the command was EXECUTE :ORCA:MYEXEC, then {Parameters} equals :ORCA:MYEXEC. The {Command} variable is set to the value of the command name.
{Prompt}	When the shell's command line editor is ready for a command line, it prints a # character as a prompt. If the {Prompt} shell variable is set to any value except the # character, the shell will print the value of the {Prompt} shell variable instead of the # character. If the {Prompt} shell variable is set to #, the shell does not print a prompt at all.
{Separator}	Under ProDOS, full path names started with the / character, and directories within path names were separated from each other, from volume names, and from file names by the / character. In GS/OS, both the / character and the : character can be used as a separator when you enter a path name, but the : character is universally used when writing a path name. If you set the Separator shell variable to a single character,

	that character will be used as a separator whenever the shell writes a path name. Note that, while many utilities make shell calls to print path names, not all do, and if the utility does not use the shell or check the {Separator} shell variable, the path names will not be consistent.
{Status}	The error status returned by the last command or EXEC file executed. This variable is the ASCII character 0 ($30) if the command completed successfully. For most commands, if an error occurred, the error value returned by the command is the ASCII string 65535 (representing the error code $FFFF).

Logical Operators

ORCA includes two operators that you can use to form boolean (logical) expressions. String comparisons are case sensitive if {CaseSensitive} is not null (the default is for string comparisons to *not* be case sensitive). If an expression result is true, then the expression returns the character 1. If an expression result is not true, then the expression returns the character 0. There must be one or more spaces before and after the comparison operator.

str1 == *str2*	String comparison: true if string *str1* and string *str2* are identical; false if not.
str1 != *str2*	String comparison: false if string *str1* and string *str2* are identical; true if not.

Operations can be grouped with parentheses. For example, the following expression is true if one of the expressions in parentheses is false and one is true; the expression is false if both expressions in parentheses are true or if both are false:

```
IF ( COWS == KINE ) != ( CATS == DOGS )
```

Every symbol or string in a logical expression must be separated from every other by at least one space. In the preceding expression, for example, there is a space between the string comparison operator != and the left parentheses, and another space between the left parentheses and the string CATS.

Entering Comments

To enter a comment into an EXEC file, start the line with an asterisk (*). The asterisk is actually a command that does nothing, so you must follow the asterisk by at least one space. For example, the following EXEC file sends a catalog listing to the printer:

```
CATALOG >.PRINTER
* Send a catalog listing to the printer
```

Use a semicolon followed by an asterisk to put a comment on the same line as a command:

```
CATALOG >.PRINTER   ;* Send a catalog listing to the printer
```

Redirecting Input and Output

Standard input is usually through the keyboard, although it can also be from a text file or the output of a program; standard output is usually to the screen, though it can be redirected to a printer or another program or disk file. You can redirect standard input and output for any command by using the following conventions on the command line:

<*inputdevice*	Redirect input to be from *inputdevice.*
>*outputdevice*	Redirect output to go to *outputdevice.*
>>*outputdevice*	Append output to the current contents of *outputdevice.*

The input device can be the keyboard or any text or source file. To redirect input from the keyboard, use the device name .CONSOLE.

The output device can be the screen, the printer, or any file. If the file named does not exist, ORCA opens a file with that name. To redirect output to the screen, use the device name .CONSOLE; to redirect output to the printer, use .PRINTER. .PRINTER is a RAM based device driver; see the section describing .PRINTER, later in this chapter, for details on when .PRINTER can be used, how it is installed, and how you can configure it.

Both input and output redirection can be used on the same command line. The input and output redirection instructions can appear in any position on the command line. For example, to redirect output from a compile of the program MYPROG to the printer, you could use either of the following commands:

```
COMPILE  MYPROG  >.PRINTER
COMPILE  >.PRINTER  MYPROG
```

To redirect output from the CATALOG command to be appended to the data already in a disk file named CATSN.DOGS, use the following command:

```
CATALOG  >>CATSN.DOGS
```

Input and output redirection can be used in EXEC files. When output is redirected when the EXEC file is executed, input and output can still be redirected from individual commands in the EXEC file.

The output of programs that do not use standard output, and the input of programs that do not use standard input, cannot be redirected.

Error messages also normally go to the screen. They can be redirected independently of standard output. To redirect error output, use the following conventions on the command line:

>&*outputdevice*	Redirect error output to go to *outputdevice.*
>>&*outputdevice*	Append error output to the current contents of *outputdevice.*

Error output devices follow the same conventions as those described above for standard output. Error output redirection can be used in EXEC files.

The .PRINTER Driver

The operating system on the Apple IIGS gives you a number of ways to write to a printer, but none of them can be used with input and output redirection, nor can they be used with standard file write commands, which is the way you would write text to a printer on many other computers. On the other hand, GS/OS does allow the installation of custom drivers, and these custom drivers can, in fact, be used with I/O redirection, and you can use GS/OS file output commands to write to a custom driver. Our solution to the problem of providing easy to use text output to a printer is to add a custom driver called .PRINTER.

As described in the last section, you can redirect either standard out or error out to your printer by using the name .PRINTER as the destination file, like this:

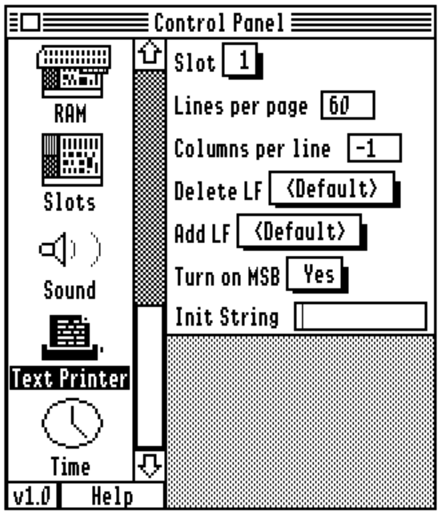
```
TYPE  MyFile  >.Printer
```

You can also open a file, using .PRINTER as the file name, using standard GS/OS calls. When you write to this file, the characters appear on your printer, rather than being written to disk. In short, as far as your programs are concerned, .PRINTER is just a write-only file.

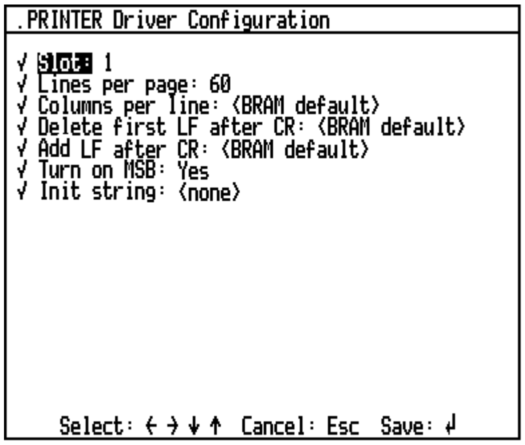
The only thing you have to watch out for is that, since .PRINTER is a RAM based driver, it must be installed on your boot disk before you can use the driver. If you are running from the system disk we sent with ORCA/Pascal, the .PRINTER driver is already installed, and you can use it right away. If you are booting from some other disk, you will need to install the .PRINTER driver on that disk. There is an installer script that will move the correct file for you, or you can simply copy the files ORCA.PRINTER and PRINTER.CONFIG from the SYSTEM:DRIVERS folder of the ORCA system disk to the SYSTEM:DRIVERS folder of your system disk.

All printers are not created equal, so any printer driver must come with some method to configure the driver. By default, our printer driver is designed to handle a serial printer installed in slot 1. It prints a maximum of 80 characters on one line, after which it will force a new line, and put any remaining characters on the new line. After printing 60 lines, a form feed is issued to advance the paper to the start of a new page. When a new line is needed, the driver prints a single carriage return character ($0D). If any of these options are unsuitable for your printer, you can change them using either a CDev or a CDA. Both of these programs produce a configuration file called PInit.Options, which will be placed in your System folder, so you need to be sure your boot disk is in a drive and not write protected when you configure your printer. This file is read by an init called TextPrinterInit at boot time to configure the text printer driver, which is itself a GS/OS driver called TextPrinter.

Figure 8.4 shows the screens you will see when you use the CDev from Apple's Control panel or when you select the CDA from the CDA menu. The options that you can select are the same for both configuration programs; these are described in Table 8.5.



Slot 1
Lines per page 60
Columns per line -1
Delete LF ⟨Default⟩
Add LF ⟨Default⟩
Turn on MSB Yes
Init String



```
.PRINTER Driver Configuration

√ Slot: 1
√ Lines per page: 60
√ Columns per line: <BRAM default>
√ Delete first LF after CR: <BRAM default>
√ Add LF after CR: <BRAM default>
√ Turn on MSB: Yes
√ Init string: <none>

     Select: ← → ↓ ↑  Cancel: Esc  Save: ↲
```

Figure 8.4: Text Printer Configuration Screens

Option	Description
Slot	This entry is the physical slot where your printer is located.
Lines per page	This entry is a single number, telling the printer driver how many lines appear on a sheet of paper. Most printers print 66 lines on a normal letter-size sheet of paper; it is traditional to print on 60 of those lines and leave the top and bottom 3 lines blank to form a margin. When the printer driver finishes printing the number of lines you specify, it issues a form-feed character ($0C), which causes most printers to skip to the top of a new page. If you set this value to 0, the printer driver will never issue a form-feed character.
Columns per line	This option is a single number telling the printer driver how many columns are on a sheet of paper. Most printers print 80 columns on a normal letter-size sheet of paper. If you use a value of -1, the printer driver will never split a line. (Using the CDA configuration program, the value before 0 shows up as BRAM default; you can use the normal control panel printer configuration page to set the line length to unlimited.) What your printer does with a line that is too long is something you would have to determine be trial and error.
Delete LF	Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. Some programs write a carriage-return line-feed combination, while

others only write a carriage-return. This option lets you tell the printer driver to strip a line-feed character if it comes right after a carriage-return character, blocking extra line-feed characters coming in from programs that print both characters.

You can select three options here: Yes, No, or BRAM Default. The Yes option strips extra line-feeds, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.

Add LF Some printers need a carriage-return line-feed character sequence to get to the start of a new line, while others only need a carriage-return. This option lets you tell the printer driver to add a line-feed character after any carriage-return character that is printed.

You can select three options here: Yes, No, or BRAM Default. The Yes option adds a line-feeds, while the No option does not. The BRAM Default option tells the printer driver to use whatever value is in the BRAM; this is the same value you would have selected using the printer configuration program in the control panel.

Turn on MSB

This line is a flag indicating whether the printer driver should set the most significant bit when writing characters to the printer. If this value is Yes the printer driver will set the most significant bit on all characters before sending the characters to the printer. If you code any number other than 0, the most significant bit will be cleared before the character is sent to the printer.

Init string This option sets a printer initialization string. This string is sent to the printer when the driver is used for the first time. With most printers and interface cards, there is some special code you can use to tell the printer that the characters that follow are special control codes. These codes are often used to control the character density, number of lines per page, font, and so forth. This initialization string, sent to the printer by the .PRINTER driver the first time the printer is used, is the traditional way of setting up your favorite defaults.

You will find many cases when you will need to send a control character to the printer as part of this initialization string. To do that using the CDev configuration program precede the character with a ~ character. For example, an escape character is actually a control-[, so you could use ~[to send an escape character to the printer. The printer driver does not do any error checking when you use the ~ character, it simply subtracts $40 from the ASCII code for the character that follows the ~ character, and sends the result to the printer. For example, g is not a control character, but ~g would still send a value, $27, to the printer. From the CDA configuration program, just type the control character in the normal way; it will show up as an inverse character on the display.

That manual that comes with your printer should have a list of the control codes you can use to configure the printer.

Table 8.5: Text Printer Configuration Options

The .PRINTER driver is a copyrighted program. If you would like to send it out with your own programs, refer to Appendix D for licensing details. (Licensing is free, but you need to include our copyright message.)

The .NULL Driver

The .NULL driver is a second driver available from GS/OS once it is installed from ORCA. This driver is primarily used in shell scripts in situations where a shell program or command is writing output you don't want to see on the screen while the script runs. In that case, you can redirect the output to .NULL. The .NULL driver does nothing with the character, so the characters are effectively ignored by the system.

Pipelines

ORCA lets you automatically execute two or more programs in sequence, directing the output of one program to the input of the next. The output of each program but the last is written to a temporary file in the work subdirectory named SYSPIPE*n*, where *n* is a number assigned by ORCA. The first temporary file opened is assigned an *n* of 0; if a second SYSPIPE*n* file is opened for a given pipeline, then it is named SYSPIPE1, and so forth.

To *pipeline*, or sequentially execute programs PROG0, PROG1, and PROG2, use the following command:

```
PROG0|PROG1|PROG2
```

The output of PROG0 is written to SYSPIPE0; the input for PROG1 is taken from SYSPIPE0, and the output is written to SYSPIPE1. The input for PROG2 is taken from SYSPIPE1, and the output is written to standard output.

SYSPIPE*n* files are text files and can be opened by the editor.

For example, if you had a utility program called UPPER that took characters from standard input, converted them to uppercase, and wrote them to standard output, you could use the following command line to write the contents of the text file MYFILE to the screen as all uppercase characters:

```
TYPE  MYFILE|UPPER
```

To send the output to the file MYUPFILE rather than to the screen, use the following command line:

```
TYPE  MYFILE|UPPER  >MYUPFILE
```

The SYSPIPE*n* files are not deleted by ORCA after the pipeline operation is complete; thus, you can use the editor to examine the intermediate steps of a pipeline as an aid to finding errors. The next time a pipeline is executed, however, any existing SYSPIPE*n* files are overwritten.

The Command Table

The command table is an ASCII text file, which you can change with the editor, or replace entirely. It is named SYSCMND, and located in the SHELL prefix of your ORCA program disk. The format of the command table is very simple. Each line is either a comment line or a command definition. Comment lines are blank lines or lines with a semicolon (;) in column one. Command lines have four fields: the command name, the command type, the command or language number, and a comment. The fields are separated by one or more blanks or tabs. The first field is the name of the command. It can be any legal GS/OS file name. Prefixes are not allowed. The second field is the command type. This can be a C (built-in command), U (utility), or L (language). The third field of a built-in command definition is the command number; the third field of a language is its language number; utilities do not use the third field. An optional comment field can follow any command.

Built-in commands are those that are predefined within the command processor, like the CATALOG command. Being able to edit the command table means that you can change the name of these commands, add aliases for them, or even remove them, but you cannot add a built-in command. As an example, UNIX fans might like to change the CATALOG command to be LS. You would do this by editing the command table. Enter LS as the command name, in column one. Enter a C, for built-in command, in column two. Enter the command number 4, obtained from looking at the command number for CATALOG in the command table, in column three. Exit the editor, saving the modified SYSCMND file. Reload the new command table by rebooting or by issuing the COMMANDS command.

Languages define the languages available on the system. You might change the language commands by adding a new language, like ORCA/Pascal. The first field contains the name of the EXE file stored in the LANGUAGES subdirectory of your ORCA system. The second field is the letter L, and the third the language number. The L can be preceded by an asterisk, which indicates that the assembler or compiler can be restarted. That is, it need not be reloaded from disk every time it is invoked. The ORCA/Pascal compiler, linker, and editor can all be restarted.

The last type of command is the utility. Utilities are easy to add to the system, and will therefore be the most commonly changed item in the command table. The first field contains the name of the utility's EXE file stored in the UTILITIES subdirectory of your ORCA system. The second field is a U. The third field is not needed, and is ignored if present. As with languages, utilities that can be restarted are denoted in the command table by preceding the U with an asterisk. Programs that can be restarted are left in memory after they have been executed. If they are called again before the memory they are occupying is needed, the shell does not have to reload the file from disk. This can dramatically increase the performance of the system. Keep in mind that not all programs can be restarted! You should not mark a program as one that can be restarted unless you are sure that it really can be safely restarted.

As an example of what has been covered so far, the command table shipped with the system is shown in Table 8.6.

```
;
;  ORCA Command Table
;
ALIAS             C        40       alias a command
ASM65816 *L       3               65816 assembler
ASML              C        1        assemble and link
ASMLG             C        2        assemble, link and execute
ASSEMBLE C        3               assemble
BREAK             C        25       break from loop
CAT               C        4        catalog
CATALOG           C        4        catalog
CC                *L       8        ORCA/C compiler
CHANGE            C        20       change language stamp
CMPL              C        1        compile and link
CMPLG             C        2        compile, link and execute
COMMANDS C        35              read command table
COMPACT           *U                compact OMF files
COMPILE           C        3        compile
COMPRESS C        32              compress/alphabetize directories
CONTINUE C        26              continue a loop
COPY              C        5        copy files/directories/disks
CREATE            C        6        create a subdirectory
CRUNCH            *U                combine object modules
DELETE            C        7        delete a file
DEREZ             *U                resource decompiler
DEVICES           C        48       Show Devices
DISABLE           C        8        disable file attributes
DISKCHECK         U                 check integrity of ProDOS disks
DUMPOBJ           U                 object module dumper
EDIT              *C       9        edit a file
ECHO              C        29       print from an exec file
ELSE              C        31       part of an IF statement
ENABLE            C        10       enable file attributes
END               C        23       end an IF, FOR, or LOOP
ENTAB             *U                entab utility
ERASE             C        44       Erase entire volume.
EXEC              L        6        EXEC language
EXECUTE           C        38       EXEC with changes to local variables
EXISTS            C        19       see if a file exists
EXIT              C        27       exit a loop
EXPORT            C        36       export a shell variable
EXPRESS           U                 converts files to ExpressLoad format
FILETYPE C        21              change the type of a file
FOR               C        22       for loop
GSBUG             U                 application version of debugger
HELP              C        11       online help
HISTORY           C        39       display last 20 commands
HOME              C        43       clear the screen and home the cursor
IF                C        30       conditional branch
INIT              C        45       initialize disks
INPUT             C        13       read a value from the command line
LINK              *C       12       link
LINKER            *L       265      command line linker script
LOOP              C        24       loop statement
MACGEN            U                 generate a macro file
MAKEBIN           U                 convert load file to a binary file
MAKELIB           U                 librarian
MOVE              C        34       move files
PASCAL            *L       5        Pascal compiler
PREFIX            C        14       set system prefix
PRIZM             U                 desktop development system
PRODOS            L        0        ProDOS language
QUIT              C        15       exit from ORCA
RENAME            C        16       rename files
```

RESEQUAL	*U		compares resource forks
REZ	*L	21	resource compiler
RUN	C	2	compile, link and execute
SET	C	28	set a variable
SHOW	C	17	show system attributes
SWITCH	C	33	switch order of files
SHUTDOWN	C	47	shut down the computer
TEXT	L	1	Text file
TOUCH	C	46	Update date/time
TYPE	C	18	list a file to standard out
UNALIAS	C	41	delete an alias
UNSET	C	37	delete a shell variable
*	C	42	comment

Table 8.6 System Commands

Command And Utility Reference

Each of the commands and utilities than ship with ORCA/Pascal are listed in alphabetic order. The syntax for the command is given, followed by a description and any parameters using the following notation:

UPPERCASE Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The shell is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.

italics Italics indicate a variable, such as a file name or address.

directory This parameter indicates any valid directory path name or partial path name. It does *not* include a file name. If the volume name is included, *directory* must start with a slash (/) or colon (:); if *directory* does not start with one of these characters, then the current prefix is assumed. For example, if you are copying a file to the subdirectory SUBDIRECTORY on the volume VOLUME, then the *directory* parameter would be: :VOLUME:SUBDIRECTORY. If the current prefix were :VOLUME:, then you could use SUBDIRECTORY for *pathname* .

The device numbers .D1, .D2, ...D*n* can be used for volume names; if you use a device name, do not precede it with a slash. For example, if the volume VOLUME in the above example were in disk drive .D1, then you could enter the *directory* parameter as .D1:SUBDIRECTORY.

GS/OS device names can be used for the volume names. Device names are the names listed by the SHOW UNITS command; they start with a period. You should not precede a device name with a slash.

GS/OS prefix numbers can be used for directory prefixes. An asterisk (*) can be used to indicate the boot disk. Two periods (..) can be used to indicate one subdirectory above the current subdirectory. If

	you use one of these substitutes for a prefix, do not precede it with a slash. For example, the HELP subdirectory on the ORCA disk can be entered as 6:HELP.
filename	This parameter indicates a file name, *not* including the prefix. The device names .CONSOLE and .PRINTER can be used as file names. Other character devices can also be used as file names, but a block device (like the name of a disk drive) cannot be used as a file name.
pathname	This parameter indicates a full path name, including the prefix and file name, or a partial path name, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory DIRECTORY on the volume VOLUME, then the *pathname* parameter would be: :VOLUME:DIRECTORY:FILE. If the current prefix were :VOLUME:, then you could use DIRECTORY:FILE for *pathname* . A full path name (including the volume name) must begin with a slash (/) or colon (:); do *not* precede *pathname* with a slash if you are using a partial path name. Character device names, like .CONSOLE and .PRINTER, can be used as file names; the device numbers .D1, .D2, ...D*n* can be used for volume names; GS/OS device names can be used a volume names; and GS/OS prefix numbers, an asterisk (*), or double periods (..) can be used instead of a prefix.
|	A vertical bar indicates a choice. For example, +L|-L indicates that the command can be entered as either +L or as -L.
A |<u>B</u>	An underlined choice is the default value.
[ ]	Parameters enclosed in square brackets are optional.
...	Ellipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

ALIAS

ALIAS [*name* [*string*]]

The ALIAS command allows you to create new commands based on existing ones. It creates an alias called *name*, which can then be typed from the command line as if it were a command. When you type the name, the command processor substitutes *string* for the name before trying to execute the command.

For example, let's assume you dump hexadecimal files with the DUMPOBJ file fairly frequently. Remembering and typing the three flags necessary to do this can be a hassle, so you might use the ALIAS command to define a new command called DUMP. The command you would use would be

```
ALIAS  DUMP  DUMPOBJ -F +X -H
```

Now, to dump MYFILE in hexadecimal format, type

```
DUMP  MYFILE
```

You can create a single alias that executes multiple commands by enclosing a string in quotes. For example,

```
ALIAS GO "CMPL MYFILE.ASM; FILETYPE MYFILE S16; MYFILE"
```

creates a new command called GO. This new command compiles and links a program, changes the file type to S16, and then executes the program.

The name and string parameters are optional. If a name is specified, but the string is omitted, the current alias for that name will be listed. If both the name and the string are omitted, a list of all current aliases and their values is printed.

Aliases are automatically exported from the LOGIN file to the command level. This means that any aliases created in the LOGIN file are available for the remainder of the session, or until you specifically delete or modify the alias. Aliases created in an EXEC file are available in that EXEC file and any other it calls, but not to the command level. See the EXECUTE command for a way to override this.

See the UNALIAS command for a way to remove an alias.

ASM65816

```
ASM65816
```

This language command sets the shell default language to 65816 Assembly Language.

While you can set the language and create assembly language files, you will not be able to assemble them unless you purchase the ORCA/M macro assembler and install it with ORCA/Pascal.

ASML

```
ASML   [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
       [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
       [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
       [language2=(option ...) ...]]
```

This internal command assembles (or compiles) and links a source file. The ORCA shell checks the language of the source file and calls the appropriate assembler or compiler. If the maximum error level returned by the assembler or compiler is less than or equal to the maximum allowed (0 unless you specify otherwise with the MERR directive or its equivalent in the source file), then the resulting object file is linked.

You can use APPEND directives (or the equivalent) to tie together source files written in different computer languages; ORCA compilers and assemblers check the language type of each file and return control to the shell when a different language must be called.

Not all compilers or assemblers make use of all the parameters provided by this command (and the ASSEMBLE, ASMLG, COMPILE, CMPL, CMPLG, and RUN commands, which use the same parameters). The ORCA/Pascal compiler, for example, includes no language-specific options, and so makes no use of the *language*=(*option* ...) parameter. If you include a parameter that a compiler or assembler cannot use, it ignores it; no error is generated. If you used append statements to tie together source files in more than one language, then all parameters are passed to every compiler, and each compiler uses those parameters that it recognizes.

Command-line parameters (those described here) override source-code options when there is a conflict.

+D|-D	+D causes debug code to be generated so that the source-level debugger may be used later when debugging the program. -D, the default, causes debug code to not be generated.
+E|-E	When a terminal error is encountered during a compile from the command line, the compiler aborts and enters the editor with the cursor on the offending line, and the error message displayed in the editor's information bar. From an EXEC file, the default is to display the error message and return to the shell. The +E flag will cause the compiler to abort to the editor, while the -E flag causes the compiler to abort to the shell.
-I	When the ORCA/C compiler compiles a program, it normally creates a .sym file in the same location as the original source file. This flag tells the compiler not to create a .sym file, and to ignore any existing .sym file. This flag is not used by the ORCA/Pascal compiler.
+L|-L	If you specify +L, the assembler or compiler generates a source listing; if you specify -L, the listing is not produced. The L parameter in this command overrides the LIST directive in the source file. +L will cause the linker to produce a link map.

+M|-M +M causes any object modules produced by the assembler or compiler to be written to memory, rather than to disk.

+O|-O ORCA/Pascal is an optimizing compiler. This flag can turn the optimizations on or off from the command line. Unlike the other parameters, and optimize pragma in the source file will override this flag.

+P|-P The compiler, linker, and many other languages print progress information as the various subroutines are processed. The -P flag can be used to suppress this progress information.

-R ORCA/C can detect changes in the source file or object files that would make it necessary to rebuild the .sym file. This flag bypasses the automatic check, forcing the compiler to rebuild the .sym file. ORCA/Pascal does not use this flag.

+S|-S If you specify +S, the linker produces an alphabetical listing of all global references in the object file; the assembler or compiler may also produce a symbol table, although the ORCA/Pascal compiler does not. If you specify -S, these symbol tables are not produced. The S parameter in this command overrides the SYMBOL directive in the source file.

+T|-T The +T flag causes all errors to be treated as terminal errors, aborting the compile. This is normally used in conjunction with +E. In that case, any error will cause the compiler to abort and enter the editor with the cursor on the offending line, and the error message displayed in the editor's information bar.

+W|-W Normally, the compiler continues compiling a program after an error has been found. If the +W flag is specified, the assembler or compiler will stop after finding an error, and wait for a keypress. Pressing . will abort the compile, entering the editor with the cursor on the offending line. Press any other key to continue the compile.

sourcefile The full path name or partial path name (including the file name) of the source file.

KEEP=*outfile* You can use this parameter to specify the path name or partial path name (including the file name) of the output file. For a one-segment program, ORCA names the object file *outfile*.ROOT. If the program contains more than one segment, ORCA places the first segment in *outfile*.ROOT and the other segments in *outfile*.A. If this is a partial compile (or several source files with different programming languages are being compiled), then other file name extensions may be used. If the compilation is followed by a successful link, then the load file is named *outfile.*

This parameter has the same effect as placing a KEEP pragma in your source file. If you have a KEEP pragma in the source file and you also use the KEEP parameter, this parameter has precedence.

When specifying a KEEP parameter, you can use two metacharacters to modify the KEEP name. If the % character is found in the keep name, the source file name is substituted. If $ is encountered, the source file name with the last extension removed is substituted.

Note the following about the KEEP parameter:

- If you use neither the KEEP parameter, the {KeepName} variable, nor the KEEP directive, then the object files are not saved at all. In this case, the link cannot be performed, because there is no object file to link.

- The file name you specify in *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and GS/OS does not allow file names longer than 15 characters.

- By default, PRIZM uses $ as the keep name. When you are using PRIZM, do not specify the keep name any other way unless in agrees with the keep name PRIZM will generate by default.

NAMES=(*seg1 seg2 ...*) This parameter causes the assembler or compiler to perform a partial assembly or compile; the operands *seg1*, *seg2*, ... specify the names of the subroutines to be assembled or compiled. In the case of objects, the object type name is given, and all of the methods within that object are recompiled. Separate the names with one or more spaces. The ORCA Linker automatically selects the latest version of each subroutine when the program is linked.

The object file created when you use the NAMES parameter contains only the specified subroutines.

You must use the same output file name for every partial compilation or assembly of a program. For example, if you specify the output file name as OUTFILE for the original compile of a program, then the compiler creates object files named OUTFILE.ROOT and OUTFILE.A. In this case you must also specify the output file name as OUTFILE for the partial compile. The new output file is named OUTFILE.B, and contains only the functions listed with the NAMES parameter. When you link a program, the linker scans all the files whose file names are identical except for their extensions, and takes the latest version of each segment.

No spaces are permitted immediately before or after the equal sign in this parameter.

language1=(*option* ...) ... This parameter allows you to pass parameters directly to specific compilers and assemblers running under the ORCA shell. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as defined in the command table), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual; the ORCA shell does no error checking on this string, but passes it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if that language compiler is not called, then the string is ignored.

No spaces are permitted immediately before or after the equal sign in this parameter.

The ORCA/Pascal compiler does not use this parameter.

ASMLG

```
ASMLG  [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
       [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
       [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
       [language2=(option ...) ...]]
```

This internal command assembles (or compiles), links, and runs a source file. Its function is identical to that of the ASML command, except that once the file has been successfully linked, it is executed automatically. See the ASML command for a description of the parameters.

ASSEMBLE

```
ASSEMBLE   [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P]
           [-R] [+S|-S] [+T|-T] [+W|-W] sourcefile
           [KEEP=outfile]
           [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
           [language2=(option ...) ...]]
```

This internal command assembles (or compiles) a source file. Its function is identical to that of the ASML command, except that the ASSEMBLE command does not call the linker to link the object files it creates; therefore, no load file is generated. You can use the LINK command to link the object files created by the ASSEMBLE command. See the ASML command for a description of the parameters.

BREAK

```
BREAK
```

This command is used to terminate a FOR or LOOP statement. The next statement executed will be the one immediately after the END statement on the closest nested FOR or LOOP statement. For example, the EXEC file

```
FOR I IN 1 2 3
  FOR J IN 2 3
    IF {I} == {J}
      BREAK
    END
    ECHO {I}
  END
END
```

would print

```
1
1
3
```

to the screen. This order results from the fact that BREAK exits from the closest loop, the `FOR J IN 2 3`, not from all loops.

CAT

```
CAT  [-A] [-D] [-H] [-L] [-N] [-P] [-T]
     [directory1 [directory2 ...]]
```

```
CAT [-A] [-D] [-H] [-L] [-N] [-P] [-T] [pathname1 [pathname2 ...]]
```

This internal command is an alternate name for CATALOG.

CATALOG

```
CATALOG  [-A] [-D] [-H] [-L] [-N] [-P] [-T]
         [directory1 [directory2 ...]]

CATALOG  [-A] [-D] [-H] [-L] [-N] [-P] [-T]
         [pathname1 [pathname2 ...]]
```

This internal command lists to standard output the directory of the volume or subdirectory you specify. More than one directory or subdirectory can be listed to get more than one catalog from a single command.

-A	GS/OS supports a status bit called the invisible bit. Finder droppings files, for example, are normally flagged as invisible so they won't clutter directory listings. The CATALOG command does not normally display invisible files when you catalog a directory; if you use the -A flag, the CATALOG command will display invisible files.
-D	If the -D flag is used, this command does a recursive catalog of directories, showing not only the directory name, but the contents of the directory, and the contents of directories contained within the directory.
-H	When this flag is used, the CATALOG command does not print the header, which shows the path being cataloged, or the trailer, which shows statistics about disk use.
-L	The standard format for a directory listing is a table, with one line per file entry. When this flag is used, the CATALOG command shows a great deal more information about each file, but the information is shown using several lines.
-N	This flag causes the CATALOG command to show only the name of the file, omitting all other information. Files are formatted with multiple file names per line, placing the file names on tab stops at 16 character boundaries. The resulting table is considerably easier to scan when looking for a specific file.
-P	The name of a file is normally displayed as a simple file name. Use of the -P flag causes the files to be listed as full path names. This option does make the file names fairly long, so the default tabular format may become cumbersome. Using this option with -L or -N clears up the problem.
-T	Most file types have a standard 3-letter identifier that is displayed by the catalog command. For example, an ASCII file has a 3-letter code of TXT. These 3-letter codes are displayed by the CATALOG command. If you use the -T flag, the CATALOG command displays the hexadecimal file type instead of the 3-letter file type code.

This flag also controls the auxiliary file type field, which is shown as a language name for SRC files. When the -T flag is used, this field, too, is shown as a hexadecimal value for all file types.

directory The path name or partial path name of the volume, directory, or subdirectory for which you want a directory listing. If the prefix is omitted, then the contents of the current directory are listed.

pathname The full path name or partial path name (including the file name) of the file for which you want directory information. You can use wildcard characters in the file name to obtain information about only specific files.

```
:ORCA.DISASM:=

Name            Type  Blocks  Modified        Created         Access  Subtype

Desktop.DISASM  S16+     230 14 Aug 90       21 May 90        DNBWR   $DB03
DISASM          EXE      101 15 Aug 90       15 Aug 90        DNBWR   $0100
DISASM.Config   $5A+       2 17 May 90       30 Apr 90        DNBWR   $800A
DISASM.Data     TXT       95 10 Aug 90       20 Oct 88        DNBWR   $0000
DISASM.Scripts  SRC       94 23 May 90       15 Aug 89        DNBWR   $0116
Help            DIR        1 18 Sep 89       14 Sep 89        DNBWR   $0000
Samples         DIR        1 13 Aug 90       14 Sep 89        DNBWR   $0000
Icons           DIR        1 17 Sep 89       14 Sep 89        DNBWR   $0000

Blocks Free:   1026      Blocks used:    574      Total Blocks:   1600
```

Table 8.7 Sample CATALOG Listing

Table 8.7 shows the output from cataloging the ORCA/Disassembler 1.2 disk. This particular disk has a good variety of file types and so forth; we'll use it to see what the CATALOG command can tell us about a disk.

The first line shows the path being cataloged; in this case, we are cataloging all files on the disk ORCA.DISASM. The last line gives more information about the disk, including the number of blocks that are not used, the number that are used, and the total number of blocks on the disk. For ProDOS format disks, a block is 512 bytes, so this disk is an 800K disk.

Between these two lines is the information about the files on the disk. The first column is the file name. If the file name is too long to fit in the space available, the rest of the information will appear on the line below.

Next is the type of the file. Most file types have a three letter code associated with them, like S16 (System 16) for a file that can be executed from the Finder or the ORCA shell, and DIR (directory) for a folder. There is no three letter code for a file with a type of $5A, so this file type is shown as the hexadecimal number for the file type. If a file is an extended file (i.e., if it has a resource fork), the file type is followed by a + character.

The column labeled "Blocks" shows the number of blocks occupied by the file on the disk. GS/OS is clever about the way it stores files, not using a physical disk block for a file that contains only zeros, for example, and programs are not necessarily loaded all at once, so this block size does not necessarily correspond to the amount of memory that will be needed to load a file or run a program; it only tells how much space is required on the disk.

The columns labeled "Modified" and "Created" give the date and time when the file was last changed and when the file was originally created, respectively. In this example, the time fields have been artificially set to 00:00 (something the Byte Works does for all of its distribution disks). When the time is set to 00:00, it is not shown.

The column labeled Access shows the values of six flags that control whether a file can be deleted (D), renamed (N), whether it has been backed up since the last time it was modified (B), whether it can be written to (W) or read from (R), and whether it is invisible (I). In all cases, if the condition is true, the flag is shown as an uppercase letter, and if the condition is false, the flag is not shown at all.

The last column, labeled "Subtype", shows the auxiliary file type for the file. For most files, this is shown as a four-digit hexadecimal number, but for SRC files you will see the name of the language.

```
Name         : Desktop.DISASM
Storage Type : 5
File Type    : S16        $B3
Aux Type     : $DB03
Access       : DNBWR      $E3
Mod Date     : 14 Aug 90
Create Date  : 21 May 90
Blocks Used  : 139
Data EOF     : $00011A6B
Res. Blocks  : 91
Res. EOF     : $0000B215
```

Table 8.8

The tabular form used by the CATALOG command to show information about files is compact, but doesn't provide enough room to show all of the information about a file that is available from GS/OS. When the -L flag is used, the CATALOG command uses an expanded form to show more information about the file. Table 8.8 shows the expanded information for the Desktop.DISASM file. The name, file type, auxiliary file type, access, modification date and creation date fields are the same as before, although the order has changed and the fields that have a hexadecimal equivalent are shown using both forms. The old block count field has been expended, showing the number of blocks used by the date fork (the Blocks Used field) and the resource fork (labeled Res. Blocks) as two separate values. In addition, the true size of the file in bytes is shown, again split between the data fork and resource fork, as the Date EOF field and the Res. EOF field. Finally, the internal storage type used by GS/OS is listed.

For a more complete and technical description of the various information returned by the CATALOG command, see *Apple IIGS GS/OS Reference*, Volume 1.

CC

```
CC
```

This language command sets the shell default language to CC, the language stamp used by the ORCA/C compiler.

While you can set the language and create C source files, you will not be able to compile them unless you purchase the ORCA/C compiler and install it with ORCA/Pascal.

CHANGE

CHANGE [-P] *pathname language*

This internal command changes the language type of an existing file.

-P	When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.
pathname	The full path name or partial path name (including the file name) of the source file whose language type you wish to change. You can use wildcard characters in the file name.
language	The language type to which you wish to change this file.

In ORCA, each source or text file is assigned the current default language type when it is created. When you assemble or compile the file, ORCA checks the language type to determine which assembler, compiler, linker, or text formatter to call. Use the CATALOG command to see the language type currently assigned to a file. Use the CHANGE command to change the language type of any of the languages listed by the SHOW LANGUAGES command.

You can use the CHANGE command to correct the ORCA language type of a file if the editor was set to the wrong language type when you created the file, for example. Another use of the CHANGE command is to assign the correct ORCA language type to an ASCII text file (GS/OS file type $04) created with another editor.

CMPL

```
CMPL   [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
       [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
       [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
       [language2=(option ...) ...]]
```

This internal command compiles (or assembles) and links a source file. Its function and parameters are identical to those of the ASML command.

CMPLG

```
CMPLG  [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
       [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
       [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
       [language2=(option ...) ...]]
```

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters.

COMMANDS

```
COMMANDS pathname
```

This internal command causes ORCA to read a command table, resetting all the commands to those in the new command table.

pathname	The full path name or partial path name (including the file name) of the file containing the command table.

When you load ORCA, it reads the command-table file named SYSCMND in prefix 15. You can use the COMMANDS command to read in a custom command table at any time. Command tables are described in the section “Command Types and the Command Table” in this chapter.

The COMMANDS command has one other useful side effect. Any program that can be restarted that has been loaded and left in memory will be purged, thus freeing a great deal of memory.

COMPACT

```
COMPACT infile [-O outfile] [-P] [-R] [-S]
```

This external command converts a load file from an uncompacted form to a compacted form.

infile	Input load file. Any OMF format file is acceptable, but the only files that benefit from the COMPACT utility are the executable files, such as EXE and S16.
-O *outfile*	By default, the input file is replaced with the compacted version of the same file. If you supply an output file name with this option, the file is written to *outfile*.

-P	When the -P flag is used, copyright and progress information is written to standard out.
-R	The -R option marks any segment named ~globals or ~arrays as a reload segment. It also forces the bank size of the ~globals segment to $10000. These options are generally only used with APW C programs.
-S	The -S flag causes a summary to be printed to standard out. This summary shows the total number of segments in the file, the number of each type of OMF record compacted, copied, and created. This information gives you some idea of what changes were made to make the object file smaller.

Compacted object files are smaller and load faster than uncompacted load files. The reduction in file size is generally about 40%, although the actual number can vary quite a bit in practice. In addition, if the original file is in OMF 1.0 format, it is converted to OMF 2.0.

Files created with ORCA/Pascal are compacted by default. The main reason for using this utility is to convert any old programs you may obtain to the newer OMF format, and to reduce their file size.

COMPILE

```
COMPILE [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
        [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
        [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
        [language2=(option ...) ...]]
```

This internal command compiles (or assembles) a source file. Its function is identical to that of the ASML command, except that it does not call the linker to link the object files it creates; therefore, no load file is generated. You can use the LINK command to link the object files created by the COMPILE command. See the ASML command for a description of the parameters.

COMPRESS

```
COMPRESS A | C | A C  [directory1 [directory2 ...]]
```

This internal command compresses and alphabetizes directories. More than one directory can be specified on a single command line.

A	Use this parameter to alphabetize the file names in a directory. The file names appear in the new sequence whenever you use the CATALOG command.
C	Use this parameter to compress a directory. When you delete a file from a directory, a “hole” is left in the directory that GS/OS fills with the file entry for

the next file you create. Use the C parameter to remove these holes from a directory, so that the name of the next file you create is placed at the end of the directory listing instead of in a hole in the middle of the listing.

A C	You can use both the A and C parameters in one command; if you do so, you must separate them with one or more spaces.
directory	The path name or partial path name of the directory you wish to compress or alphabetize, *not* including any file name. If you do not include a volume or directory path, then the current directory is acted on.

This command works only on GS/OS directories, not on other file systems such as DOS or Pascal. Due to the design of GS/OS, the COMPRESS command will also not work on the disk volume that you boot from – to modify the boot volume of your hard disk, for example, you would have to boot from a floppy disk.

To interchange the positions of two files in a directory, use the SWITCH command.

CONTINUE

```
CONTINUE
```

This command causes control to skip over the remaining statements in the closest nested FOR or LOOP statement. For example, the EXEC file

```
FOR I
  IF {I} == IMPORTANT
    CONTINUE
  END
  DELETE {I}
END
```

would delete all files listed on the command line when the EXEC file is executed except for the file IMPORTANT.

COPY

```
COPY [-C] [-F] [-P] [-R] pathname1 [pathname2]
COPY [-C] [-F] [-P] [-R] pathname1 [directory2]
COPY directory1 directory2
COPY [-D] volume1 volume2
```

This internal command copies a file to a new subdirectory, or to a duplicate file with a different file name. This command can also be used to copy an entire directory or to perform a block-by-block disk copy.

-C	If you specify -C before the first path name, COPY does not prompt you if the target file name (*pathname2*) already exists.
-D	If you specify -D before the first path name, both path names are volume names, and both volumes are the same size, then a block-by-block disk copy is performed. Other flags, while accepted, are ignored when this flag is used.
-F	Normally, the COPY command copies both the data fork and the resource fork of a file. When the -F flag is used, only the data fork is copied. If the destination file already exists, it's resource fork is left undisturbed. By copying the data fork of a file onto an existing file with a resource fork, it is possible to combine the data fork of the original file with the resource fork of the target file.
-P	The COPY command prints progress information showing what file is being copied as it works through a list of files. The -P flag suppresses this progress information.
-R	Normally, the COPY command copies both the data fork and the resource fork of a file. When the -R flag is used, only the resource fork is copied. If the destination file already exists, it's data fork is left undisturbed. By copying the resource fork of a file onto an existing file with a data fork, it is possible to add the resource fork of the original file to the data fork of the target file.
pathname1	The full or partial path name (including the file name) of a file to be copied. Wildcard characters may be used in the file name.
pathname2	The full or partial path name (including the file name) to be given to the copy of the file. Wildcard characters can *not* be used in this file name. If you leave this parameter out, then the current directory is used and the new file has the same name as the file being copied.
directory1	The path name or partial path name of a directory that you wish to copy. The entire directory (including all the files, subdirectories, and files in the subdirectories) is copied.
directory2	The path name or partial path name of the directory to which you wish to copy the file or directory. If *directory2* does not exist, it is created (unless *directory1* is empty). If you do not include this parameter, the current directory is used.
volume1	The name of a volume that you want to copy onto another volume. The entire volume (including all the files, subdirectories, and files in the subdirectories) is copied. If both path names are volume names, both volumes are the same size, *and* you specify the -D parameter, then a block-by-block disk copy is performed. You can use a device name (such as .D1) instead of a volume name.

volume2 The name of the volume that you want to copy onto. You can use a device name instead of a volume name.

If you do not specify *pathname2*, and a file with the file name specified in *pathname1* exists in the target subdirectory, or if you do specify *pathname2* and a file named *pathname2* exists in the target subdirectory, then you are asked if you want to replace the target file. Type Y and press RETURN to replace the file. Type N and press RETURN to copy the file to the target prefix with a new file name. In the latter case, you are prompted for the new file name. Enter the file name, or press RETURN without entering a file name to cancel the copy operation. If you specify the -C parameter, then the target file is replaced without prompting.

If you do not include any parameters after the COPY command, you are prompted for a path name, since ORCA prompts you for any required parameters. However, since the target prefix and file name are not required parameters, you are *not* prompted for them. Consequently, the current prefix is always used as the target directory in such a case. To copy a file to any subdirectory *other* than the current one, you *must* include the target path name as a parameter either in the command line or following the path name entered in response to the file name prompt.

If you use volume names for both the source and target and specify the -D parameter, then the COPY command copies one volume onto another. In this case, the contents of the target disk are destroyed by the copy operation. The target disk must be initialized (use the INIT command) *before* this command is used. This command performs a block-by-block copy, so it makes an exact duplicate of the disk. Both disks must be the same size and must be formatted using the same FST for this command to work. You can use device names rather than volume names to perform a disk copy. To ensure safe volume copies, it is a good idea to write-protect the source disk.

CREATE

CREATE *directory1* [*directory2* ...]

This internal command creates a new subdirectory. More than one subdirectory can be created with a single command by separating the new directory names with spaces.

directory The path name or partial path name of the subdirectory you wish to create.

CRUNCH

CRUNCH [-P] *rootname*

This external command combines the object files created by partial assemblies or compiles into a single object file. For example, if a compile and subsequent partial compiles have produced the object files FILE.ROOT, FILE.A, FILE.B, and FILE.C, then the CRUNCH command combines FILE.A, FILE.B, and FILE.C into a new file called FILE.A, deleting the old object files in the process. The new FILE.A contains only the latest version of each function in the program. New functions added during partial compiles are placed at the end of the new FILE.A.

-P	Suppresses the copyright and progress information normally printed by the CRUNCH utility.
rootname	The full path name or partial path name, including the file name but minus any file name extensions, of the object files you wish to compress. For example, if your object files are named FILE.ROOT, FILE.A, and FILE.B in subdirectory :HARDISK:MYFILES:, you should then use :HARDISK:MYFILES:FILE for *rootname.*

DELETE

```
DELETE [-C] [-P] [-W] pathname1 [pathname2 ...]
```

This internal command deletes the file you specify. You can delete more than one file with a single command by separating multiple file names with spaces.

-C	If you delete the entire contents of a directory by specifying = for the path name, or if you try to delete a directory, the DELETE command asks for confirmation before doing the delete. If you use the -C flag, the delete command does not ask for confirmation before doing the delete.
-P	When you delete files using wildcards, or when you delete a directory that contains other files, the delete command lists the files as they are deleted. To suppress this progress information, use the -P flag.
-W	When you try to delete a file that does not exist, the DELETE command prints a warning message, but does not flag an error by returning a non-zero status code. If you use the -W flag, the warning message will not be printed.
pathname	The full path name or partial path name (including the file name) of the file to be deleted. Wildcard characters may be used in the file name.

If the target file of the DELETE command is a directory, the directory and all of its contents, including any included directories and their contents, are deleted.

DEREZ

```
DEREZ   [-D[EFINE] macro[=data]] [-E[SCAPE]] [-I pathname]
        [-M[AXTRINGSIZE] n] [-O filename]
        [-ONLY typeexpr[(id1[:id2])]] [-P] [-RD]
        [-S[KIP] typeexpr[(id1[:id2])]] [-U[NDEF] macro]
        resourceFile [resourceDescriptionFile]
```

This external command reads the resource fork of an extended file, writing the resources in a text form. This output is detailed enough that it is possible to edit the output, then recompile it with the Rez compiler to create a new, modified resource fork.

`-D[EFINE]` *`macro`*`[=`*`data`*`]` Defines the macro *macro* with the value *data*. This is completely equivalent to placing the statement

#define *macro data*

at the start of the first resource description file.

If the optional data field is left off, the macro is defined with a null value.

More than one -d option can be used on the command line.

`-E[SCAPE]` Characters outside of the range of the printing ASCII characters are normally printed as escape sequences, like \0xC1. If the -e option is used, these characters are sent to standard out unchanged. Not all output devices have a mechanism defined to print these characters, so using this option may give strange or unusable results.

`-I` *`pathname`* Lets you specify one or more path names to search for #include files. This option can be used more than once. If the option is used more than once, the paths are searched in the order listed.

`-M[AXTRINGSIZE]` *`n`* This setting controls the width of the output. It must be in the range 2 to 120.

`-O` *`filename`* This option provides another way of redirecting the output. It should not be used if command line output redirection is also used. With the -O option, the file is created with a file type of SRC and a language type of Rez.

`-ONLY` *`typeexpr`*`[(`*`id1`*`[:`*`id2`*`])]` Lists only resources with a resource type of *typeexpr*, which should be expressed as a numeric value. If the value is followed immediately (no spaces!) by a resource ID number in parenthesis, only that particular resource is listed. To list a range of resources, separate the starting and ending resource ID with a colon.

`-P` When this option is used, the copyright, version number, and progress information is written to standard out.

`-RD` Suppresses warning messages if a resource type is redeclared.

`-S[KIP]` *`typeexpr`*`[(`*`id1`*`[:`*`id2`*`])]` Lists all but the resources with a resource type of *typeexpr*, which should be expressed as a numeric value. If the value is followed immediately (no spaces!) by a resource ID number in parenthesis, only that particular resource is skipped. To skip a range of resources, separate the starting and ending resource ID with a colon.

`-U[NDEF]` *`macro`* This option can be used to undefine a macro variable.

`resourceFile` This is the name of the extended file to process. The resource fork from this file is converted to text form and written to standard out.

`resourceDescriptionFile` This file contains a series of declarations in the same format as used by the Rez compiler. More than one resource description file can be used. Any include (not #include), read, data, and resource statements are skipped, and the remaining declarations are used as format specifiers, controlling how DeRez writes information about any particular resource type.

If no resource description file is given, or if DeRez encounters a resource type for which none of the resource description files provide a format, DeRez writes the resource in a hexadecimal format.

The output from DeRez consists of resource and data statements that are acceptable to the Rez resource compiler. If the output from DeRez is used immediately as the input to the resource compiler, the resulting resource fork is identical to the one processed by DeRez. In some cases, the reverse is not true; in particular, DeRez may create a data statement for some input resources.

Numeric values, such as the argument for the -only option, can be listed as a decimal value, a hexadecimal value with a leading $, as in the ORCA assembler, or a hexadecimal value with a leading 0x, as used by the Pascal language.

For all resource description files specified on the source line, the following search rules are applied:

1. DeRez tries to open the file as is, by appending the file name given to the current default prefix.
2. If rule 1 fails and the file name contains no colons and does not start with a colon (in other words, if the name is truly a file name, and not a path name or partial path name), DeRez appends the file name to each of the path names specified by -i options and tries to open the file.
3. DeRez looks for the file in the folder 13:RInclude.

For more information about resource compiler source files and type declarations, see Chapter 10.

DEVICES

```
DEVICES [-B] [-D] [-F] [-I] [-L] [-M] [-N] [-S] [-T] [-U] [-V]
```

The DEVICES command lists all of the devices recognized by GS/OS in a tabular form, showing the device type, device name, and volume name. Various flags can be used to show other information about the devices in an expanded form.

-B	Display the block size for block devices.
-D	Display the version number of the software driver for the device.
-F	Show the number of free blocks remaining on a block device.
-I	Display the file system format used by the device.
-L	Show all available information about each device. This would be the same as typing all of the other flags.
-M	Show the total number of blocks on the device.
-N	Display the device number.
-S	Display the slot number of the device.
-T	Show the type of the device.
-U	Show the unit number for the device.
-V	Show the volume name for the device.

The name of the device is always displayed, but when you use any flag except -L, the device type and volume name are not shown unless you specifically use the -T and -V flags.

See the GS/OS Technical Reference Manual for a detailed description of what devices are, and what the various fields mean in relation to any particular device.

DISABLE

DISABLE [-P] D | N | B | W | R | I *pathname*

This internal command disables one or more of the access attributes of a GS/OS file.

-P	When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.
D	"Delete" privileges. If you disable this attribute, the file cannot be deleted.
N	"Rename" privileges. If you disable this attribute, the file cannot be renamed.
B	"Backup required" flag. If you disable this attribute, the file will not be flagged as having been changed since the last time it was backed up.
W	"Write" privileges. If you disable this attribute, the file cannot be written to.
R	"Read" privileges. If you disable this attribute, the file cannot be read.
I	"Visible" flag. If you disable this attribute, the file will be displayed by the CATALOG command without using the -A flag. In other words, invisible files become visible.
pathname	The full path name or partial path name (including the file name) of the file whose attributes you wish to disable. You can use wildcard characters in the file name.

You can disable more than one attribute at one time by typing the operands with no intervening spaces. For example, to "lock" the file TEST so that it cannot be written to, deleted, or renamed, use the command

```
DISABLE DNW TEST
```

Use the ENABLE command to reenable attributes you disabled with the DISABLE command.

When you use the CATALOG command to list a directory, the attributes that are currently enabled are listed in the access field for each file.

DISKCHECK

DISKCHECK *volume|device*

This external command scans the disk for active files and lists all block allocations, including both data and resource forks of any extended file types. It will then notify you of block conflicts, where two or more files are claiming the same block(s), and provide an opportunity to list the blocks and files involved. Finally, it will verify the integrity of the disk's bitmap. Bitmap errors will be reported and you can choose to repair the bitmap.

`volume|device` The GS/OS volume name or device name of the disk to check. The volume name can be specified with or without a beginning colon or slash; for example,

```
DiskCheck :HardDisk
DiskCheck HardDisk
```

A device name requires a period before the name; for example, .SCSI1. Volume numbers can also be used, as in .D2.

DISKCHECK will only verify a ProDOS volume. It will not work with an HFS volume.

In normal display mode, data scrolls continuously on the screen. While DISKCHECK is running, press the space bar to place DISKCHECK in single step mode. In this mode, block allocations are displayed one at a time, each time the space bar is pressed. Press return to return to normal display mode.

DISKCHECK will check volumes with up to 65535 blocks of 512 bytes (32M).

DISKCHECK makes the following assumptions:

- Blocks zero and one are always used and contain boot code.
- Enough disk integrity exists to make a GetFileInfo call on the volume.
- Block two is the beginning of the volume directory and contains valid information regarding the number of blocks, bitmap locations, entries per block, and entry size.
- All unused bytes at the end of the last bitmap block are truly unused; that is, they will be set to zero whenever the bitmap is repaired.

DISKCHECK may not catch invalid volume header information as an error. Likewise, DISKCHECK does not check all details of the directory structures. Therefore, if large quantities of errors are displayed, it is likely that the volume header information or directory information is at fault.

ECHO

`ECHO [-N] [-T] *string*`

This command lets you write messages to standard output. All characters from the first non-blank character to the end of the line are written to standard out. You can use redirection to write the characters to error out or a disk file.

-N The -N flag suppresses the carriage return normally printed after the string, allowing other output to be written to the same line. One popular use for this option is to write a prompt using the ECHO command, then use the INPUT command to read a value. With the -N flag, the input cursor appears on the same line as the prompt.

-T	By default, and tab characters in the string are converted to an appropriate number of spaces before the string is written. If the -T flag is used, the tab characters are written as is.
string	The characters to write.

If you want to start your string with a space or a quote mark, enclose the string in quote marks. Double the quote marks to imbed a quote in the string. For example,

```
ECHO "   This string starts with 3 spaces and includes a "" character."
```

EDIT

`EDIT` *`pathname1 pathname2 ...`*

This external command calls the ORCA editor and opens a file to edit.

pathname1	The full path name or partial path name (including the file name) of the file you wish to edit. If the file named does not exist, a new file with that name is opened. If you use a wildcard character in the file name, the first file matched is opened. If more than one file name is given, up to ten files are opened at the same time.

The ORCA default language changes to match the language of the open file. If you open a new file, that file is assigned the current default language. Use the CHANGE command to change the language stamp of an existing file. To change the ORCA default language before opening a new file, type the name of the language you wish to use, and press RETURN.

The editor is described in Chapter 9.

ELSE

`ELSE`

`ELSE IF` *`expression`*

This command is used as part of an IF command.

ENABLE

```
ENABLE  [-P] D |  N | B | W | R | I pathname
```

This internal command enables one or more of the access attributes of a GS/OS file, as described in the discussion of the DISABLE command. You can enable more than one attribute at one time by typing the operands with no intervening spaces. For example, to "unlock" the file TEST so that it can be written to, deleted, or renamed, use the command

```
ENABLE  DNW  TEST
```

When a new file is created, all the access attributes are enabled. Use the ENABLE command to reverse the effects of the DISABLE command. The parameters are the same as those of the DISABLE command.

ENTAB

```
ENTAB [-L language] [file]
```

This external command scans a text stream, converting runs of tabs and space characters into the minimum number of tabs and space characters needed to present the same information on the display screen. Tabs are not used to replace runs of spaces in quoted strings.

-L *language*	The ENTAB utility checks the language stamp of the input file and uses the appropriate tab line from the SYSTABS file to determine the location of tab stops. This flag can be used to override the default language number, forcing the utility to use the tab line for some other language. You can use either a language number or a language name as the parameter.
file	File to process.

There is no DETAB utility, but the TYPE command can be used to strip tab characters from a file, replacing the tab characters with an appropriate number of space characters.

END

```
END
```

This command terminates a FOR, IF, or LOOP command.

ERASE

```
ERASE [-C] device [name]
```

This internal command writes the initialization tracks used by GS/OS to a disk that has already been formatted as a GS/OS disk. In effect, this erases all files on the disk.

-C	Normally, the system will ask for permission (check) before erasing a disk. The -C flag disables that check.
device	The device name (such as .D1) of the disk drive containing the disk to be formatted; or, if the disk being formatted already has a volume name, you can specify the volume name instead of a device name.
name	The new volume name for the disk. If you do not specify *name*, then the name :BLANK is used.

ORCA recognizes the device type of the disk drive specified by *device*, and uses the appropriate format. ERASE works for all disk formats supported by GS/OS.

ERASE destroys any files on the disk being formatted. The effect of the ERASE command is very similar to the effect of the INIT command, but there are some differences. The INIT command will work on any disk, while the ERASE command can only be used on a disk that has already been initialized. The ERASE command works much faster than the INIT command, since the ERASE command does not need to take the time to create each block on the disk. Finally, when the INIT command is used, each block is filled with zeros. The ERASE command does not write zeros to the existing blocks, so any old information on the disk is not truly destroyed; instead, it is hidden very, very well, just as if all of the files and folders on the disk had been deleted.

EXEC

```
EXEC
```

This language command sets the shell default language to the EXEC command language. When you type the name of a file that has the EXEC language stamp, the shell executes each line of the file as a shell command.

EXECUTE

`EXECUTE` *`pathname`* `[`*`paramlist`*`]`

This internal command executes an EXEC file. If this command is executed from the ORCA Shell command line, then the variables and aliases defined in the EXEC file are treated as if they were defined on the command line.

pathname	The full or partial path name of an EXEC file. This file name cannot include wildcard characters.
paramlist	The list of parameters being sent to the EXEC file.

EXISTS

`EXISTS` *`pathname`*

This internal command checks to see if a file exists. If the file exists, the {Status} shell variable is set to 1; if the file does not exist, the {Status} shell variable is set to 0. Several disk related errors can occur, so be sure to check specifically for either a 0 or 1 value. When using this command in an EXEC file, keep in mind that a non-zero value for the {Status} variable will cause an EXEC file to abort unless the {Exit} shell variable has been cleared with an UNSET EXIT command.

pathname	The full or partial path name of a file. More than one file can be checked at the same time by specifying multiple path names. In this case, the result is zero only if each and every file exists.

EXIT

`EXIT [`*`number`*`]`

This command terminates execution of an EXEC file. If *number* is omitted, the {Status} variable will be set to 0, indicating a successful completion. If *number* is coded, the {Status} variable will be set to the number. This allows returning error numbers or condition codes to other EXEC files that may call the one this statement is included in.

number	Exit error code.

EXPORT

`EXPORT [`*`variable1`*` [`*`variable2`*` ...]]`

This command makes the specified variable available to EXEC files called by the current EXEC file. When used in the LOGIN file, the variable becomes available at the command level, and in all EXEC files executed from the command level. More than one variable may be exported with a single command by separating the variable names with spaces.

*variable*n	Names of the variables to export.

EXPRESS

`EXPRESS [-P]` *`infile`* `-O` *`outfile`*

The external command EXPRESS reformats an Apple IIGS load file so that it can be loaded by the ExpressLoad loader that comes with Apple's system disk, starting with version 5.0 of the system disk. When loaded with ExpressLoad, the file will load much faster than it would load using the standard loader; however, files reformatted for use with ExpressLoad can still be loaded by the System Loader.

`-P`	If you specify this option, EXPRESS displays progress information. If you omit it, progress information is not displayed.
`infile`	The full or partial path name of a load file.
`-O` *`outfile`*	This is the full or partial path name of the file to write. Unlike many commands, this output file is a required parameter.

Since the linker that comes with ORCA can automatically generate a file that is expressed, this utility is generally only used to reformat executable programs you obtain through other sources.

EXPRESS only accepts version 2.0 OMF files as input. You can check the version number of the OMF file using DUMPOBJ, and convert OMF 1.0 files to OMF 2.0 using COMPACT.

ExpressLoad does not support multiple load files; therefore, you cannot use Express with any program that references segments in a run-time library.

The following system loader calls are not supported by ExpressLoad:

- GetLoadSegInfo ($0F) The internal data structures of ExpressLoad are not the same as those of the System Loader.
- LoadSegNum ($0B) Because EXPRESS changes the order of the segments in the load file, an application that uses this call and has been converted by EXPRESS cannot be processed by the System Loader. Use the LoadSegName function instead.

- UnloadSegNum ($0C) Because EXPRESS changes the order of the segments in the load file, an application that uses this call and has been converted by EXPRESS cannot be processed by the System Loader. Use the UnloadSeg ($0E) function instead.

FILETYPE

```
FILETYPE [-P] pathname filetype [auxtype]
```

This internal command changes the GS/OS file type, and optionally the auxiliary file type, of a file.

-P	When wildcards are used, a list of the files changed is written to standard out. The -P flag suppresses this progress information.
pathname	The full path name or partial path name (including the file name) of the file whose file type you wish to change.
filetype	The GS/OS file type to which you want to change the file. Use one of the following three formats for *filetype*:

- A decimal number 0-255.
- A hexadecimal number $00-$FF.
- The three-letter abbreviation for the file type used in disk directories; for example, S16, OBJ, EXE. A partial list of GS/OS file types is shown in Table 8.13.

auxtype	The GS/OS auxiliary file type to which you want to change the file. Use one of the following two formats for *auxtype*:

- A decimal number 0-65535.
- A hexadecimal number $0000-$FFFF.

You can change the file type of any file with the FILETYPE command; ORCA does not check to make sure that the format of the file is appropriate. However, the GS/OS call used by the FILETYPE command may disable some of the access attributes of the file. Use the CATALOG command to check the file type and access-attribute settings of the file; use the ENABLE command to reenable any attributes that are disabled by GS/OS.

The linker can automatically set the file type and auxiliary file type of a program.

Decimal	Hex	Abbreviation	File Type
001	$01	BAD	Bad blocks file
002	$02	PCD	Pascal code file (SOS)
003	$03	PTX	Pascal text file (SOS)
004	$04	TXT	ASCII text file
005	$05	PDA	Pascal data file (SOS)
006	$06	BIN	ProDOS 8 binary load
007	$07	FNT	Font file (SOS)
008	$08	FOT	Graphics screen file
009	$09	BA3	Business BASIC program file (SOS)
010	$0A	DA3	Business BASIC data file (SOS)
011	$0B	WPF	Word processor file (SOS)
012	$0C	SOS	SOS system file (SOS)
015	$0F	DIR	Directory
016	$10	RPD	RPS data file (SOS)
017	$11	RPI	RPS index file (SOS)
176	$B0	SRC	Source
177	$B1	OBJ	Object
178	$B2	LIB	Library
179	$B3	S16	GS/OS system file
180	$B4	RTL	Run-time library
181	$B5	EXE	Shell load file
182	$B6	STR	load file
184	$B8	NDA	New desk accessory
185	$B9	CDA	Classic desk accessory
186	$BA	TOL	Tool file
200	$C8	FNT	Font file
226	$E2	DTS	Defile RAM tool patch
240	$F0	CMD	ProDOS CI added command file
249	$F9	P16	ProDOS 16 file
252	$FC	BAS	BASIC file
253	$FD	VAR	EDASM file
254	$FE	REL	REL file
255	$FF	SYS	ProDOS 8 system load file

Table 8.13. A Partial List of GS/OS File Types

FOR

`FOR` *`variable`* `[IN` *`value1 value2`* `... ]`

This command, together with the END statement, creates a loop that is executed once for each parameter value listed. Each of the parameters is separated from the others by at least on space. To include spaces in a parameter, enclose it in quote marks. For example, the EXEC file

```
FOR I IN GORP STUFF "FOO BAR"
   ECHO {I}
END
```

would print

```
GORP
STUFF
FOO BAR
```

to the screen.

If the IN keyword and the strings that follow are omitted, the FOR command loops over the command line inputs, skipping the command itself. For example, the EXEC file named EXECFILE

```
FOR I
   ECHO {I}
END
```

would give the same results as the previous example if you executed it with the command

```
EXECFILE GORP STUFF "FOO BAR"
```

HELP

`HELP [`*`commandname1`*` [`*`commandname2`*` ...]]`

This internal command provides on-line help for all the commands in the command table provided with the ORCA development environment. If you omit *commandname*, then a list of all the commands in the command table are listed on the screen.

commandname The name of the ORCA shell command about which you want information.

When you specify *commandname*, the shell looks for a text file with the specified name in the HELP subdirectory in the UTILITIES prefix (prefix 17). If it finds such a file, the shell prints the contents of the file on the screen. Help files contain information about the purpose and use of commands, and show the command syntax in the same format as used in this manual.

If you add commands to the command table, or change the name of a command, you can add, copy, or rename a file in the HELP subdirectory to provide information about the new command.

HISTORY

```
HISTORY
```

This command lists the last twenty commands entered in the command line editor. Commands executed in EXEC files are not listed.

HOME

```
HOME
```

This command sends a $0C character to the standard output device. The output can be redirected to files, printers, or error output using standard output redirection techniques.

When the $0C character is sent to the console output device, the screen is cleared and the cursor is moved to the top left corner of the screen. When the $0C character is sent to most printers, the printer will skip to the top of the next page.

IF

`IF` *`expression`*

This command, together with the ELSE IF, ELSE, and END statements provides conditional branching in EXEC files. The expression is evaluated. If the resulting string is the character 0, the command interpreter skips to the next ELSE IF, ELSE or END statement, and does not execute the commands in between. If the string is anything but the character 0, the statements after the IF statement are executed. In that case, if an ELSE or ELSE IF is encountered, the command skips to the END statement associated with the IF.

The ELSE statement is used to provide an alternate set of statements that will be executed if the main body of the IF is skipped due to an expression that evaluates to 0. It must appear after all ELSE IF statements.

ELSE IF is used to test a series of possibilities. Each ELSE IF clause is followed by an expression. If the expression evaluates to 0, the statements following the ELSE IF are skipped; if the expression evaluates to anything but 0, the statements after the ELSE IF are executed.

As an example, the following code will translate an Arabic digit (contained in the variable {I}) into a Roman numeral.

```
IF {I} == 1
   ECHO I
ELSE IF {I} == 2
   ECHO II
ELSE IF {I} == 3
   ECHO III
ELSE IF {I} == 4
   ECHO IV
ELSE IF {I} == 5
   ECHO V
ELSE
   ECHO The number is too large for this routine.
END
```

INIT

`INIT [-C]` *`device`* `[`*`fst`*`] [`*`name`*`]`

This external command formats a disk as a GS/OS volume.

-C	Disable checking. If the disk has been previously initialized, the system will ask for permission (check) before starting initialization. The default is to check.
device	The device name (such as .D1) of the disk drive containing the disk to be formatted; or, if the disk being formatted already has a volume name, you can specify the volume name instead of a device name.
fst	The file system translator number. The default FST is 1 (ProDOS).
name	The new volume name for the disk. If you do not specify *name*, then the name :BLANK is used.

ORCA recognizes the device type of the disk drive specified by *device*, and uses the appropriate format. INIT works for all disk formats supported by GS/OS.

GS/OS is capable of supporting a wide variety of physical disk formats and operating system file formats. The term file system translator, or FST, has been adopted to refer to the various formats. By default, when you initialize a disk, the INIT command uses the physical format and operating system format that has been in use by the ProDOS and GS/OS operating system since ProDOS was introduced for the Apple //e computer. If you would like to use a different FST, you can specify the FST as a decimal number. Apple has defined a wide variety of numbers for use as FSTs, although there is no reason to expect that all of them will someday be implemented in GS/OS; some of the FST numbers are shown in Table 8.14, and a more complete list can be found in *Apple IIGS GS/OS Reference*, Volume 1. Not all of these FSTs have been implemented in GS/OS as this manual goes to press. Even if an FST has been implemented, not all FSTs can be used on all formats of floppy disks. If you aren't sure if an FST is available, give it a try – if not, you will get an error message.

INIT destroys any files on the disk being formatted.

FST Number	File System
1	ProDOS (Apple II, Apple IIGS) and SOS (Apple ///)
2	DOS 3.3
3	DOS 3.2
4	Apple II Pascal
5	Macintosh MFS
6	Macintosh HFS
7	Lisa
8	Apple CP/M
10	MS/DOS
11	High Sierra
13	AppleShare

Table 8.14 FST Numbers

INPUT

```
INPUT variable
```

This command reads a line from standard input, placing all of the characters typed, up to but not including the carriage return that marks the end of the line, in the shell variable *variable*.

variable	Shell variable in which to place the string read from standard in.

LINK

```
LINK [+B|-B] [+C|-C] [+L|-L] [+P|-P] [+S|-S] [+X|-X] objectfile
     [KEEP=outfile]

LINK [+B|-B] [+C|-C] [+L|-L] [+P|-P] [+S|-S] [+X|-X] objectfile1
     objectfile2  ... [KEEP=outfile]
```

This internal command calls the ORCA linker to link object files to create a load file. You can use this command to link object files created by assemblers or compilers, and to cause the linker to search library files.

+B|-B	The +B flag tells the linker to create a bank relative program. Each load segment in a bank relative program must be aligned to a 64K bank boundary by

the loader. When the current version of the Apple IIGS loader loads a bank relative program, it also purges virtually all purgeable memory, which could slow down operations of programs like the ORCA shell, which allows several programs to stay in memory. Bank relative programs take up less disk space than programs that can be relocated to any memory space, and they load faster, since all two-byte relocation information can be resolved at link time, rather than creating relocation records for each address.

+C|-C Executable files are normally compacted, which means some relocation information is packed into a compressed form. Compacted files load faster and use less room on disk than uncompacted files. To create an executable file that is not compacted, use the -C flag.

+L|-L If you specify +L, the linker generates a listing (called a link map) of the segments in the object file, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify -L, the link map is not produced.

+P|-P The linker normally prints a series of dots as subroutines are processed on pass one and two, followed by the length of the program and the number of executable segments in the program. The -P flag can be used to suppress this progress information.

+S|-S If you specify +S, the linker produces an alphabetical listing of all global references in the object file (called a symbol table). If you specify -S, the symbol table is not produced.

+X|-X Executable files are normally expressed, which means they have an added header and some internal fields in the code image are expanded. Expressed files load from disk faster than files that are not expressed, but they require more disk space. You can tell the linker not to express a file by using the -X flag.

objectfile The full or partial path name, minus file name extension, of the object files to be linked. All files to be linked must have the same file name (except for extensions), and must be in the same subdirectory. For example, the program TEST might consist of object files named TEST.ROOT, TEST.A, and TEST.B, all located in directory :ORCA:MYPROG:. In this case, you would use :ORCA:MYPROG:TEST for *objectfile*.

objectfile1 objectfile2,... You can link several object files into one load file with a single LINK command. Enclose in parentheses the full path names or partial path names, minus file name extensions, of all the object files to be included; separate the file names with spaces. Either a .ROOT file or a .A file must be present. For example, the program TEST might consist of object files named TEST1.ROOT, TEST1.A, TEST1.B, TEST2.A, and TEST2.B, all in directory

:ORCA:MYPROG:. In this case, you would use :ORCA:MYPROG:TEST1 for *objectfile* and :ORCA:MYPROG:TEST2 for *objectfile1*.

You can also use this command to specify one or more library files (GS/OS file type $B2) to be searched. Any library files specified are searched in the order listed. Only the segments needed to resolve references that haven't already been resolved are extracted from the standard library files.

KEEP=*outfile* Use this parameter to specify the path name or partial path name of the executable load file.

If you do not use the KEEP parameter, then the link is performed, but the load file is not saved.

If you do not include any parameters after the LINK command, you are prompted for an input file name, as ORCA prompts you for any required parameters. However, since the output path name is not a required parameter, you are *not* prompted for it. Consequently, the link is performed, but the load file is not saved. To save the results of a link, you *must* include the KEEP parameter in the command line or create default names using the {LinkName} variable.

The linker can automatically set the file type and auxiliary file type of the executable file it creates.

To automatically link a program after assembling or compiling it, use one of the following commands instead of the LINK command: ASML, ASMLG, CMPL, CMPLG.

LINKER

```
LINKER
```

This language command sets the shell default language for linker script files.

LOOP

```
LOOP
```

This command together with the END statement defines a loop that repeats continuously until a BREAK command is encountered. This statement is used primarily in EXEC files. For example, if you have written a program called TIMER that returns a {Status} variable value of 1 when a particular time has been reached, and 65535 for an error, you could cause the program SECURITY.CHECK to be executed each time TIMER returned 1, and exit the EXEC file when TIMER returned 65535. The EXEC file would be

```
UNSET EXIT
LOOP
   TIMER
   SET STAT {STATUS}
   IF {STAT} == 1
      SECURITY.CHECK
   ELSE IF {STAT} == 65535
      BREAK
   END
END
```

MAKELIB

```
MAKELIB [-F] [-D] [-P] libfile  [ + | - | ^  objectfile1
        + | - | ^ objectfile2 ...]
```

This external command creates a library file.

-F	If you specify -F, a list of the file names included in *libfile* is produced. If you leave this option out, no file name list is produced.
-D	If you specify -D, the dictionary of symbols in the library is listed. Each symbol listed is a global symbol occurring in the library file. If you leave this option out, no dictionary is produced.
-P	Suppresses the copyright and progress information normally printed by the MAKELIB utility.
libfile	The full path name or partial path name (including the file name) of the library file to be created, read, or modified.
+*objectfilen*	The full path name or partial path name (including the file name) of an object file to be added to the library. You can specify as many object files to add as you wish. Separate object file names with spaces.
-*objectfilen*	The file name of a component file to be removed from the library. This parameter is a file name only, not a path name. You can specify as many component files to remove as you wish. Separate file names with spaces.
^*objectfilen*	The full path name or partial path name (including the file name) of a component file to be removed from the library and written out as an object file. If you include a prefix in this path name, the object file is written to that prefix. You can specify as many files to be written out as object files as you wish. Separate file names with spaces.

An ORCA library file (GS/OS file type $B2) consists of one or more component files, each containing one or more segments. Each library file contains a library-dictionary segment that the linker uses to find the segments it needs.

MAKELIB creates a library file from any number of object files. In addition to indicating where in the library file each segment is located, the library-dictionary segment indicates which object file each segment came from. The MAKELIB utility can use that information to remove any component files you specify from a library file; it can even recreate the original object file by extracting the segments that made up that file and writing them out as an object file. Use the (-F) and (-D) parameters to list the contents of an existing library file.

The MAKELIB command is for use only with ORCA object-module-format (OMF) library files used by the linker. For information on the creation and use of libraries used by language compilers, consult the manuals that came with those compilers.

MAKELIB accepts either OMF 1 or OMF 2 files as input, but always produces OMF 2 files as output. MAKELIB literally converts OMF 1 files to OMF 2 files before placing them in the library. Among other things, this gives you one way to convert an OMF 1 file to an OMF 2 file: first create a library with the OMF 1 file, then extract the file from the library. The extracted file will be in OMF 2 format.

To create an OMF library file using the ORCA/Pascal compiler, use the following procedure:

1. Write one or more source files in the normal way, but don't use `main` as one of the functions.

2. Compile the programs. Each source file is saved as two object files, one with the extension .ROOT, and one with the extension .A.

3. Run the MAKELIB utility, specifying each object file to be included in the library file, but ignoring any .ROOT files. For example, if you compiled two source files, creating the object files LIBOBJ1.ROOT, LIBOBJ1.A, LIBOBJ2.ROOT, LIBOBJ2.A, and your library file is named LIBFILE, then your command line should be as follows:

   ```
   MAKELIB LIBFILE +LIBOBJ1.A +LIBOBJ2.A
   ```

4. Place the new library file in the LIBRARIES: subdirectory. (You can accomplish this in step 3 by specifying 13:LIBFILE for the library file, or you can use the MOVE command after the file is created.)

MOVE

MOVE [-C] [-P] *pathname1* [*pathname2*]

MOVE [-C] [-P] *pathname1* [*directory2*]

This internal command moves a file from one directory to another; it can also be used to rename a file.

-C	If you specify -C before the first file name, then MOVE does not prompt you if the target file name (*filename2*) already exists.
-P	The MOVE command prints progress information showing what file is being moved as it works through a list of files. The -P flag suppresses this progress information.
pathname1	The full path name or partial path name (including the file name) of the file to be moved. Wildcard characters may be used in this file name.
pathname2	The full path name or partial path name of the directory you wish to move the file to. If you specify a target file name, the file is renamed when it is moved. Wildcard characters can *not* be used in this path name. If the prefix of *pathname2* is the same as that of *pathname1,* then the file is renamed only.
directory2	The path name or partial path name of the directory you wish to move the file to. If you do not include a file name in the target path name, then the file is not renamed. Wildcard characters can *not* be used in this path name.

If *pathname1* and the target directory are on the same volume, then ORCA calls GS/OS to move the directory entry (and rename the file, if a target file name is specified). If the source and destination are on different volumes, then the file is copied; if the copy is successful, then the original file is deleted. If the file specified in *pathname2* already exists and you complete the move operation, then the old file named *pathname2* is deleted and replaced by the file that was moved.

NEWER

NEWER *pathname1 pathname2...*

This internal command checks to see if any file in a list of files has been modified since the first file was modified. If the first file is newer than, or as new as, all of the other files, the {Status} shell variable is set to 0. If any of the files after the first file is newer than the first file, the {Status} shell variable is set to 1.

pathname1	The full or partial path name of the file to be checked.
pathname2...	The full or partial path name of the files to compare with the first file. If any of the files in this list have a modification date after *pathname1*, {Status} is set to 1.

This command is most commonly used in script files to create sophisticated scripts that automatically decide when one of several files in a project need to be recompiled.

The GS/OS operating system records the modification date to the nearest minute. It is quite possible, unfortunately, to make changes to more than one file, then attempt to rebuild a file, in less than one minute. In this case, the command may miss a file that has been changed. See the TOUCH command for one way to update the time stamp.

Wildcards may be used in any path name. If the first file is specified with a wildcard, only the first matching file is checked. If wildcards are used in the remaining names, each matching file is checked against the first file.

It is possible for the NEWER command to return a value other than 0 or 1; this would happen, for example, if a disk is damaged or if one of the files does not exist at all. For this reason, your script files should check for specific values of 0 or 1.

A status variable other than zero generally causes a script file to exit. To prevent this, be sure and unset the exit shell variable.

PASCAL

```
PASCAL
```

This language command sets the shell default language to PASCAL, the language stamp used by ORCA/Pascal.

PREFIX

PREFIX [-C] [*n*] *directory*[:]

This internal command sets any of the eight standard GS/OS prefixes to a new subdirectory.

-C	The PREFIX command does not normally allow you to set a prefix to a path name that does not exist or is not currently available. The -C flag overrides this check, allowing you to set the prefix to any valid GS/OS path name.
n	A number from 0 to 31, indicating the prefix to be changed. If this parameter is omitted, 8 is used. This number must be preceded by one or more spaces.
directory	The full or partial path name of the subdirectory to be assigned to prefix *n*. If a prefix number is used for this parameter, you must follow the prefix number with the : character.

Prefix 8 is the current prefix; all shell commands that accept a path name use prefix 8 as the default prefix if you do not include a colon (:) at the beginning of the path name. Prefixes 9 through 17 are used for specific purposes by ORCA, GS/OS and the Apple IIGS tools; see the section “Standard Prefixes” in this chapter for details. The default settings for the prefixes are shown in Table 8.3. Prefixes 0 to 7 are obsolete ProDOS prefixes, and should no longer be used. Use the SHOW PREFIX command to find out what the prefixes are currently set to.

The prefix assignments are reset to the defaults each time ORCA is booted. To use a custom set of prefix assignments every time you start ORCA, put the PREFIX commands in the LOGIN file.

PRODOS

```
PRODOS
```

This language command sets the ORCA shell default language to GS/OS text. GS/OS text files are standard ASCII files with GS/OS file type $04; these files are recognized by GS/OS as text files. ORCA TEXT files, on the other hand, are standard ASCII files with GS/OS file type $B0 and an ORCA language type of TEXT. The ORCA language type is not used by GS/OS.

QUIT

```
QUIT
```

This internal command terminates the ORCA program and returns control to GS/OS. If you called ORCA from another program, GS/OS returns you to that program; if not, GS/OS prompts you for the next program to load.

RENAME

RENAME *pathname1 pathname2*

This internal command changes the name of a file. You can also use this command to move a file from one subdirectory to another on the same volume.

pathname1	The full path name or partial path name (including the file name) of the file to be renamed or moved. If you use wildcard characters in the file name, the first file name matched is used.
pathname2	The full path name or partial path name (including the file name) to which *pathname1* is to be changed or moved. You cannot use wildcard characters in the file name.

If you specify a different subdirectory for *pathname2* than for *pathname1*, then the file is moved to the new directory and given the file name specified in *pathname2.*

The subdirectories specified in *pathname1* and *pathname2* must be on the same volume. To rename a file and move it to another volume, use the MOVE command.

RESEQUAL

`RESEQUAL [-P] `*`pathname1 pathname2`*

The external command RESEQUAL compares the resources in two files and writes their differences to standard out.

RESEQUAL checks that each file contains resources of the same type and identifier as the other file; that the size of the resources with the same type and identifier are the same; and that their contents are the same.

`-P`	If this flag is used, a copyright message and progress information is written to error out.
`pathname1`	The full or partial path name of one of the two files to compare.
`pathname2`	The full or partial path name of one of the two files to compare.

If a mismatch is found, the mismatch and the subsequent 15 bytes are written to standard out. RESEQUAL then continues the comparison, starting with the byte following the last byte displayed. The following messages appear when reporting differences:

- In 1 but not in 2

  The resource type and ID are displayed.

- In 2 but not in 1

  The resource type and ID are displayed.

- Resources are different sizes

  The resource type, resource ID, and the size of the resource in each file are displayed.

- Resources have different contents

  This message is followed by the resource type and ID, then by the offset in the resource, and 16 bytes of the resource, starting at the byte that differed. If more than ten differences are found in the same resource, the rest of the resource is skipped and processing continues with the next resource.

REZ

```
REZ
```

This language command sets the default language to Rez. The resource compiler is described in Chapter 10.

RUN

```
RUN    [+D|-D] [+E|-E] [-I] [+M|-M] [+L|-L] [+O|-O] [+P|-P] [-R]
       [+S|-S] [+T|-T] [+W|-W] sourcefile  [KEEP=outfile]
       [NAMES=(seg1[ seg2[ ...]])] [language1=(option ...)
       [language2=(option ...) ...]]
```

This internal command compiles (or assembles), links, and runs a source file. Its function is identical to that of the ASMLG command. See the ASML command for a description of the parameters.

SET

```
SET [variable [value]]
```

This command allows you to assign a value to a variable name. You can also use this command to obtain the value of a variable or a list of all defined variables.

variable	The variable name you wish to assign a value to. Variable names are not case sensitive, and only the first 255 characters are significant. If you omit *variable*, then a list of all defined names and their values is written to standard output.
value	The string that you wish to assign to *variable*. Values are case sensitive and are limited to 65536 characters. All characters, including spaces, starting with the first non-space character after *variable* to the end of the line, are included in *value*. If you include *variable* but omit *value*, then the current value of *variable* is written to standard output. Embed spaces within *value* by enclosing *value* in double quote marks.

A variable defined with the SET command is normally available only in the EXEC file where it is defined, or if defined on the command line, only from the command line. The variable and its value are not normally passed on to EXEC files, nor are the variables set in an EXEC file available to the caller of the EXEC file.

To pass a variable and its value on to an EXEC file, you must export the variable using the EXPORT command. From that time on, any EXEC file will receive a copy of the variable. Note that this is a copy: UNSET commands used to destroy the variable, or SET commands used to change it, will not affect the original. Variables exported from the LOGIN file are exported to the command level.

You can cause changes to variables made in an EXEC file to change local copies. See the EXECUTE command for details.

Use the UNSET command to delete the definition of a variable.

Certain variable names are reserved; see "Programming EXEC Files, earlier in this chapter, for a list of reserved variable names.

SHOW

```
SHOW [LANGUAGE] [LANGUAGES] [PREFIX] [TIME] [UNITS]
```

This internal command provides information about the system.

LANGUAGE	Shows the current system-default language.
LANGUAGES	Shows a list of all languages defined in the language table, including their language numbers.
PREFIX	Shows the current subdirectories to which the GS/OS prefixes are set. See the section “Standard Prefixes” in this chapter for a discussion of ORCA prefixes.
TIME	Shows the current time.
UNITS	Shows the available units, including device names and volume names. Only those devices that have formatted GS/OS volumes in them are shown. To see the device names for all of your disk drives, make sure that each drive contains a GS/OS disk.

More than one parameter can be entered on the command line; to do so, separate the parameters by one or more spaces. If you enter no parameters, you are prompted for them.

SHUTDOWN

```
SHUTDOWN
```

This internal command shuts down the computer, ejecting floppy disks and leaving any RAM disk intact. A dialog will appear which allows you to restart the computer.

Technically, the command performs internal clean up of the shell's environment, just as the QUIT command does, ejects all disks, and then does an OSShutDown call with the shut down flags set to 0.

SWITCH

SWITCH [-P] *pathname1 pathname2*

This internal command interchanges two file names in a directory.

-P	When wildcards are used, the names of the two files switched are written to standard out. The -P flag suppresses this progress information.
pathname1	The full path name or partial path name (including the file name) of the first file name to be moved. If you use wildcard characters in the file name, the first file name matched is used.
pathname2	The full path name or partial path name (including the file name) to be switched with *pathname1* . The prefix in *pathname2* must be the same as the prefix in *pathname1*. You cannot use wildcard characters in this file name.

For example, suppose the directory listing for :ORCA:MYPROGS: is as follows in the figure below:

```
:ORCA:MYPROGS:=
Name         Type   Blocks     Modified           Created    Access  Subtype

C.SOURCE      SRC     5    26 MAR 86 07:43   29 FEB 86 12:34  DNBWR   C
COMMAND.FILE  SRC     1     9 APR 86 19:22   31 MAR 86 04 22  DNBWR   EXE
ABS.OBJECT    OBJ     8    12 NOV 86 15:02    4 MAR 86 14:17  NBWR
```

Figure 8.15. CATALOG :ORCA:MYPROGS: command

To reverse the positions in the directory of the last two files, use the following command:

```
SWITCH  :ORCA:MYPROGS:COMMAND.FILE  :ORCA:MYPROGS:ABS.OBJECT
```

Now if you list the directory again, it looks like this:

```
:ORCA:MYPROGS:=
Name         Type   Blocks     Modified           Created    Access  Subtype

C.SOURCE      SRC     5    26 MAR 86 07:43   29 FEB 86 12:34  DNBWR   C
ABS.OBJECT    OBJ     8    12 NOV 86 15:02    4 MAR 86 14:17   NBWR
COMMAND.FILE  SRC     1     9 APR 86 19:22   31 MAR 86 04 22  DNBWR   EXE
```

Figure 8.16. CATALOG :ORCA:MYPROGS: command

You can alphabetize GS/OS directories with the COMPRESS command, and list directories with the CATALOG command. This command works only on GS/OS directories, not on other file systems such as DOS or Pascal. Due to the design of GS/OS, the SWITCH command will also not work on the disk volume that you boot from – to modify the boot volume of your hard disk, for example, you would have to boot from a floppy disk.

TEXT

```
TEXT
```

This language command sets the ORCA shell default language to ORCA TEXT. ORCA text files are standard-ASCII files with GS/OS file type $B0 and an ORCA language type of TEXT. The TEXT file type is provided to support any text formatting programs that may be added to ORCA. TEXT files are shown in a directory listing as SRC files with a subtype of TEXT.

Use the PRODOS command to set the language type to GS/OS text; that is, standard ASCII files with GS/OS file type $04. PRODOS text files are shown in a directory listing as TXT files with no subtype.

TOUCH

TOUCH [-P] *pathname*

This internal command "touches" a file, changing the file's modification date and time stamp to the current date and time, just as if the file had been loaded into the editor and saved again. The contents of the file are not affected in any way.

-P	When wildcards are used, a list of the files touched is written to standard out. The -P flag suppresses this progress information.
pathname	The full path name or partial path name (including the file name) of the file to be touched. You can use wildcard characters in this file name, in which case every matching file is touched. You can specify more than one path name in the command; separate path names with spaces.

TYPE

TYPE [+N|<u>-N</u>] [<u>+T</u>|-T] *pathname1* [*startline1* [*endline1*]] [*pathname2*
    [*startline2* [*endline2*]]...]

This internal command prints one or more text or source files to standard output (usually the screen).

+N|-N	If you specify +N, the shell precedes each line with a line number. The default is -N: no line numbers are printed.
+T|-T	The TYPE command normally expands tabs as a file is printed; using the -T flag causes the TYPE command to send tab characters to the output device unchanged.

pathname	The full path name or partial path name (including the file name) of the file to be printed. You can use wildcard characters in this file name, in which case every text or source file matching the wildcard file name specification is printed. You can specify more than one path name in the command; separate path names with spaces.
start linen	The line number of the first line of this file to be printed. If this parameter is omitted, then the entire file is printed.
endlinen	The line number of the last line of this file to be printed. If this parameter is omitted, then the file is printed from *startline* to the end of the file.

ORCA text files, GS/OS text files, and ORCA source files can be printed with the TYPE command. Use the TYPE command and output redirection to merge files. For example, to merge the files FILE1 and FILE2 into the new file FILE3, use the command:

```
TYPE FILE1 FILE2 > FILE3
```

Normally, the TYPE command functions as a DETAB utility, expanding tabs to an appropriate number of spaces as the file it sent to the output device. The TYPE command examines the language stamp of the file being typed, reading the appropriate tab line from the SYSTABS file to determine where the tab stops are located.

If you are using the type command to append one file to the end of another, you may not want tabs to be expanded. In That case, the -T flag can be used to suppress tab expansions.

UNALIAS

`UNALIAS` *`variable1`* `[`*`variable2`* `...]`

The UNALIAS command deletes an alias created with the ALIAS command. More than one alias can be deleted by listing all of them, separated by spaces.

UNSET

`UNSET` *`variable1`* `[`*`variable2`*`...]`

This command deletes the definition of a variable. More than one variable may be deleted by separating the variable names with spaces.

variable	The name of the variable you wish to delete. Variable names are not case sensitive, and only the first 255 characters are significant.

Use the SET command to define a variable.

*

`* string`

The * command is the comment. By making the comment a command that does nothing, you are able to rename it to be anything you wish. Since it is a command, the comment character must be followed by a space. All characters from there to the end of the line, or up to a ; character, which indicates the start of the next command, are ignored.

Chapter 9 – The Text Editor

The ORCA editor allows you to write and edit source and text files. This chapter provides reference material on the editor, including detailed descriptions of all editing commands.

The first section in this chapter, “Modes,” describes the different modes in which the editor can operate. The second section, “Macros,” describes how to create and use editor macros, which allow you to execute a string of editor commands with a single keystroke. The third section, "Using Editor Dialogs," gives a general overview of how the mouse and keyboard are used to manipulate dialogs. The next section, “Commands,” describes each editor command and gives the key or key combination assigned to the command. The last section, “Setting Editor Defaults,” describes how to set the defaults for editor modes and tab settings for each language.

Modes

The behavior of the ORCA editor depends on the settings of several modes, as follows:

- Insert.
- Escape.
- Auto Indent.
- Text Selection.
- Hidden Characters.

Most of these modes has two possible states; you can toggle between the states while in the editor. The default for these modes can be changed by changing flags in the SYSTABS file; this is described later in this chapter, in the section "Setting Editor Defaults." All of these modes are described in this section.

Insert

When you first start the editor, it is in over strike mode; in this mode the characters you type replace any characters the cursor is on. In insert mode, any characters you type are inserted at the left of the cursor; the character the cursor is on and any characters to the right of the cursor are moved to the right.

The maximum number of characters the ORCA editor will display on a single line is 255 characters, and this length can be reduced by appropriate settings in the tab line. If you insert enough characters to create a line longer than 255 characters, the line is wrapped and displayed as more than one line. Keep in mind that most languages limit the number of characters on a single source line to 255 characters, and may ignore any extra characters or treat them as if they were on a new line.

To enter or leave the insert mode, type ⌘E. When you are in insert mode, the cursor will be an underscore character that alternates with the character in the file. In over strike mode, the cursor is a blinking box that changes the underlying character between an inverse character (black on white) and a normal character (white on black).

Escape

When you press the ESC key, the editor enters the escape mode. For the most part, the escape mode works like the normal edit mode. The principle difference is that the number keys allows you to enter repeat counts, rather than entering numbers into the file. After entering a repeat count, a command will execute that number of times.

For example, the B command inserts a blank line in the file. If you would like to enter fifty blank lines, you would enter the escape mode, type 50B, and leave the escape mode by typing the ESC key a second time.

Earlier, it was mentioned that the number keys were used in escape mode to enter repeat counts. In the normal editor mode, followed by a number key moves to various places in the file. In escape mode, the key modifier allows you to type numbers.

The only other difference between the two modes is the way CTRL_ works. This key is used primarily in macros. If you are in the editor mode, CTRL_ places you in escape mode. If you are in escape mode, it does nothing. In edit mode, CTRL_ does nothing; in escape mode, it returns you to edit mode. This lets you quickly get into the mode you need to be in at the start of an editor macro, regardless of the mode you are in when the macro is executed.

The remainder of this chapter describes the standard edit mode.

Auto Indent

You can set the editor so that RETURN moves the cursor to the first column of the next line, or so that it follows indentations already set in the text. If the editor is set to put the cursor on column 1 when you press RETURN, then changing this mode causes the editor to put the cursor on the first non-space character in the next line; if the line is blank, then the cursor is placed under the first non-space character in the first non-blank line above the cursor. The first mode is generally best for line-oriented languages, like assembly language or BASIC. The second is handy for block-structured languages like C or Pascal.

To change the return mode, type RETURN.

Select Text

You can use the mouse or the keyboard to select text in the ORCA editor. This section deals with the keyboard selection mechanism; see "Using the Mouse," later in this chapter, for information about selecting text with the mouse.

The Cut, Copy, Delete and Block Shift commands require that you first select a block of text. The ORCA editor has two modes for selecting text: line-oriented and character-oriented selects. As you move the cursor in line-oriented select mode, text or code is marked a line at a time. In the character-oriented select mode, you can start and end the marked block at any character. Line-oriented select mode is the default for assembly language; for text files and most high-level languages, character-oriented select mode is the default.

While in either select mode, the following cursor-movement commands are active:

- bottom of screen
- top of screen
- cursor down
- cursor up
- start of line
- screen moves

In addition, while in character-oriented select mode, the following cursor-movement commands are active:

- cursor left
- cursor right
- end of line
- tab
- tab left
- word right
- word left

As you move the cursor, the text between the original cursor position and the final cursor position is marked (in inverse characters). Press RETURN to complete the selection of text. Press ESC to abort the operation, leave select mode, and return to normal editing.

To switch between character- and line-oriented selection while in the editor, type CTRL⌘x.

Hidden Characters

There are cases where line wrapping or tab fields may be confusing. Is there really a new line, or was the line wrapped? Do those eight blanks represent eight spaces, a tab, or some combination of spaces and tabs? To answer these questions, the editor has an alternate display mode that shows hidden characters. To enter this mode, type ⌘=; you leave the mode the same way. While you are in the hidden character mode, end of line characters are displayed as the mouse text return character. Tabs are displayed as a right arrow where the tab character is located, followed by spaces until the next tab stop.

Macros

You can define up to 26 macros for the ORCA editor, one for each letter on the keyboard. A macro allows you to substitute a single keystroke for up to 128 predefined keystrokes. A macro can contain both editor commands and text, and can call other macros.

To create a macro, press ESC. The current macro definitions for A to J appear on the screen. The LEFT-ARROW and RIGHT-ARROW keys can be used to switch between the three pages of macro definitions. To replace a definition, press the key that corresponds to that macro, then type in the new macro definition. You must be able to see a macro to replace it - use the left and right arrow keys to get the correct page. Press OPTION ESC to terminate the macro definition. You can include CTRL*key* combinations, *key* combinations, OPTION*key* combinations, and the RETURN, ENTER, ESC, and arrow keys. The following conventions are used to display keystrokes in macros:

CTRL*key*	The uppercase character *key* is shown in inverse.
key	An inverse A followed by *key* (for example, AK)
OPTION*key*	An inverse B followed by *key* (for example, BK)
ESC	An inverse left bracket (CTRL [).
RETURN	An inverse M (CTRL M).
ENTER	An inverse J (CTRL J).
UP-ARROW	An inverse K (CTRL K).
DOWN-ARROW	An inverse J (CTRL J).
LEFT-ARROW	An inverse H (CTRL H).
RIGHT-ARROW	An inverse U (CTRL U).
DELETE	A block

Each *key* combination or OPTION*key* combination counts as two keystrokes in a macro definition. Although an *key* combination looks (in the macro definition) like a CTRL A followed by *key*, and an OPTION*key* combination looks like a CTRL B followed by *key*, you cannot enter CTRL A when you want an or CTRL B when you want an OPTION key.

If you make a mistake typing a macro definition, you can back up with DELETE. If you wish to retype the macro definition, press OPTION ESC to terminate the definition, press the letter key for the macro you want to define, and begin over. When you are finished entering macros, press OPTION ESC to terminate the last option definition, then press OPTION to end macro entry. If you have entered any new macro definitions, a dialog will appear asking if you want to save the macros to disk; select OK to save the new macro definitions, and Cancel to return to the editor. If you select Cancel, the macros you have entered will remain in effect until you leave the editor.

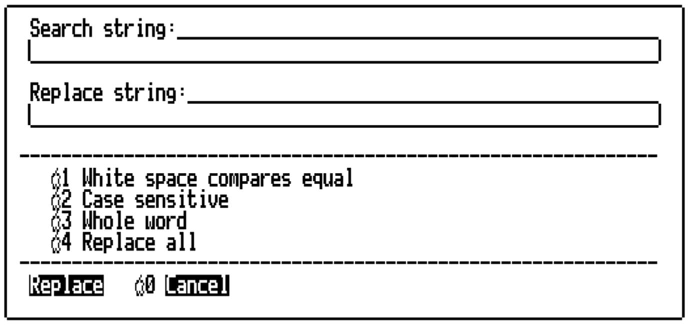
Macros are saved on disk in the file SYSEMAC in the ORCA shell prefix.

To execute a macro, hold down OPTION and press the key corresponding to that macro.

Using Editor Dialogs

The text editor makes use of a number of dialogs for operations like entering search strings, selecting a file to open, and informing you of error conditions. The way you select options, enter text, and execute commands in these dialogs is the same for all of them.

Figure 9.1 shows the Search and Replace dialog, one of the most comprehensive of all of the editor's dialogs, and one that happens to illustrate many of the controls used in dialogs.



```
Search string:
Replace string:

   ⌘1 White space compares equal
   ⌘2 Case sensitive
   ⌘3 Whole word
   ⌘4 Replace all

Replace   ⌘0 Cancel
```

Figure 9.1

The first item in this dialog is an editline control that lets you enter a string. When the dialog first appears, the cursor is at the beginning of this line. You can use any of the line editing commands from throughout the ORCA programming environment to enter and edit a string in this editline control; these line editing commands are summarized in Table 9.2.

command	command name and effect
LEFT-ARROW	**cursor left** - The cursor will move to the left.
RIGHT-ARROW	**cursor right** - The cursor will move to the right.
⌘> or ⌘.	**end of line** - The cursor will move to the right-hand end of the string.
⌘< or ⌘,	**start of line** - The cursor will move to the left-hand end of the string.
⌘Y or CTRLY	**delete to end of line** - Deletes characters from the cursor to the end of the line.
⌘Z or CTRLZ	**undo** - Resets the string to the starting string.
ESC or CTRLX	**exit** - Stops string entry, leaving the dialog without changing the default string or executing the command.
⌘E or CTRLE	**toggle insert mode** - Switches between insert and over strike mode. The dialog starts out in the same mode as the editor, but switching the mode in the dialog does not change the mode in the editor.
DELETE	**delete character left** - Deletes the character to the left of the cursor, moving the cursor left.

Table 9.2 Editline Control Commands

The Search and Replace dialog has two editline items; you can move between them using the tab key. You may also need to enter a tab character in a string, either to search specifically for a string that contains an imbedded tab character, or to place a tab character in a string that will replace the string once it is found. To enter a tab character in an editline string, use tab. While only one space will appear in the editline control, this space does represent a tab character.

Four options appear below the editline controls. Each of these options is preceded by an character and a number. Pressing x, where x is the number, selects the option, and causes a check mark to appear to the left of the option. Repeating the operation deselects the option, removing the check mark. You can also select and deselect options by using the mouse to position the cursor over the item, anywhere on the line from the character to the last character in the label.

At the bottom of the dialog is a pair of buttons; some dialogs have more than two, while some have only one. These buttons cause some action to occur. In general, all but one of these buttons will have an character and a number to the left of the button. You can select a button in one of several ways: by clicking on the button with the mouse, by pressing the RETURN key (for the default button, which is the one without an character), by pressing x, or by pressing the first letter of the label on the button. (For dialogs with an editline item, the last option is not available.)

Once an action is selected by pressing a button, the dialog will vanish and the action will be carried out.

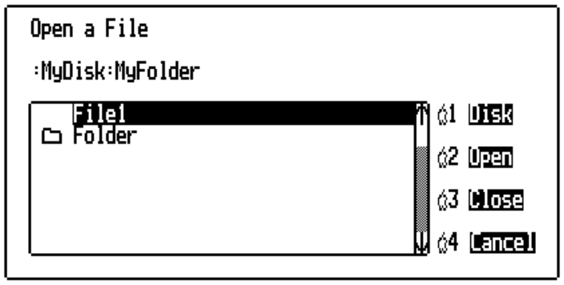


Figure 9.3

Figure 9.3 shown the Open dialog. This dialog contains a list control, used to display a list of files and folders.

You can scroll through the list by clicking on the arrows with the mouse, dragging the thumb with the mouse (the thumb is the space in the gray area between the up and down arrows), clicking in the gray area above or below the thumb, or by using the up and down arrow keys.

If there are any files in the list, one will always be selected. For commands line Open that require a file name, you will be able to select any file in the list; for commands like New, that present the file list so you know what file names are already in use, only folders can be selected. You can change which file is selected by clicking on another file or by using the up or down arrow keys. If you click on the selected name while a folder is selected, the folder is opened. If you click on a selected file name, the file is opened.

Using the Mouse

All of the features of the editor can be used without a mouse, but the mouse can also be used for a number of functions. If you prefer not to use a mouse, simply ignore it. You can even disconnect the mouse, and the ORCA editor will perform perfectly as a text-based editor.

The most common use for the mouse is moving the cursor and selecting text. To position the cursor anywhere on the screen, move the mouse. As soon as the mouse is moved, an arrow will appear on the screen; position this arrow where you would like to position the cursor and click.

Several editor commands require you to select some text. With any of these commands, you can select the text before using the command by clicking to start a selection, then dragging the mouse while holding down the button while you move to the other end of the selection. Unlike keyboard selection, mouse selections are always done in character select mode. You can also select words by double-clicking to start the selection, or lines by triple clicking to start the selection. Finally, if you drag the mouse off of the screen while selecting text, the editor will start to scroll one line at a time.

The mouse can also be used to select dialog buttons, change dialog options, and scroll list items in a dialog. See "Using Editor Dialogs" in this chapter for details.

Command Descriptions

This section describes the functions that can be performed with editor commands. The key assignments for each command are shown with the command description.

Screen-movement descriptions in this manual are based on the direction the display screen moves through the file, not the direction the lines appear to move on the screen. For example, if a command description says that the screen scrolls down one line, it means that the lines on the screen move *up* one line, and the next line in the file becomes the bottom line on the screen.

CTRL**@** — About

Shows the current version number and copyright for the editor. Press any key or click on the mouse to get rid of the About dialog.

CTRL**G** — Beep the Speaker

The ASCII control character BEL ($07) is sent to the output device. Normally, this causes the speaker to beep.

, or < **Beginning of Line**

The cursor is placed in column one of the current line.

DOWN-ARROW **Bottom of Screen / Page Down**

The cursor moves to the last visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the bottom of the screen, the screen scrolls down twenty-two lines.

CTRL**C or C** **Copy**

When you execute the Copy command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix. (To cancel the Copy operation without writing the block to SYSTEMP, press ESC instead of RETURN.) Use the Paste command to place the copied material at another position in the file.

CTRL**W or W** **Close**

Closes the active file. If the file has been changed since the last update, a dialog will appear, giving you a chance to abort the close, save the changes, or close the file without saving the changes. If the active file is the only open file, the editor exits after closing the file; if there are other files, the editor selects the next file to become the active file.

DOWN-ARROW **Cursor Down**

The cursor is moved down one line, preserving its horizontal position. If it is on the last line of the screen, the screen scrolls down one line.

LEFT-ARROW **Cursor Left**

The cursor is moved left one column. If it is in column one, the command is ignored.

RIGHT-ARROW **Cursor Right**

The cursor is moved right one column. If it is on the end-of-line marker (usually column 80), the command is ignored.

UP-ARROW — Cursor Up

The cursor is moved up one line, preserving its horizontal position. If it is on the first line of the screen, the screen scrolls up one line. If the cursor is on the first line of the file, the command is ignored.

CTRL**X** or ⌘**X** — Cut

When you execute the Cut command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use cursor-movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is written to the file SYSTEMP in the work prefix, and deleted from the file. (To cancel the Cut operation without cutting the block from the file, press ESC instead of RETURN). Use the Paste command to place the cut text at another location in the file.

⌘ESC — Define Macros

The editor enters the macro definition mode. Press OPTION ESC to terminate a definition, and OPTION to terminate macro definition mode. The macro definition process is described in the section "Macros" in this chapter.

⌘DELETE — Delete

When you execute the delete command, the editor enters select mode, as discussed in the section "Select Text" in this chapter. Use any of the cursor movement or screen-scroll commands to mark a block of text (all other commands are ignored), then press RETURN. The selected text is deleted from the file. (To cancel the delete operation without deleting the block from the file, press ESC instead of RETURN.)

CTRL**F** or ⌘**F** — Delete Character

The character that the cursor is on is deleted and put in the Undo buffer (see the description of the Undo command). Characters to the right of the cursor are moved one space to the left to fill in the gap. The last column on the line is replaced by a space.

DELETE or CTRL**D** — Delete Character Left

The character to the left of the cursor is deleted, and the character that the cursor is on, as well as the rest of the line to the right of the cursor, are moved 1 space to the left to fill in the gap. If the cursor is in column one and the over strike mode is active, no action is taken. If the cursor is in column one and the insert mode is active, then the line the cursor is on is appended to the line above and the cursor remains on the character it was on before the delete. Deleted characters are put in the undo buffer.

T or CTRLT **Delete Line**

The line that the cursor is on is deleted, and the following lines are moved up one line to fill in the space. The deleted line is put in the Undo buffer (see the description of the Undo command).

CTRLY or Y **Delete to EOL**

The character that the cursor is on, and all those to the right of the cursor to the end of the line, are deleted and put in the Undo buffer (see the description of the Undo command).

G **Delete Word**

When you execute the delete word command, the cursor is moved to the beginning of the word it is on, then delete character commands are executed for as long as the cursor is on a non-space character, then for as long as the cursor is on a space. This command thus deletes the word plus all spaces up to the beginning of the next word. If the cursor is on a space, that space and all following spaces are deleted, up to the start of the next word. All deleted characters, including spaces, are put in the Undo buffer (see the description of the Undo command).

. or > **End of Line**

If the last column on the line is not blank, the cursor moves to the last column. If the last column is blank, then the cursor moves to the right of the last non-space character in the line. If the entire line is blank, the cursor is placed in column 1.

? or / **Help**

Displays the help file, which contains a short summary of editor commands. Use ESC to return to the file being edited.

The help file is a text file called SYSHELP, found in the shell prefix. Since it is a text file, you can modify it as desired.

B or CTRLB **Insert Line**

A blank line is inserted at the cursor position, and the line the cursor was on and the lines below it are scrolled down to make room. The cursor remains in the same horizontal position on the screen.

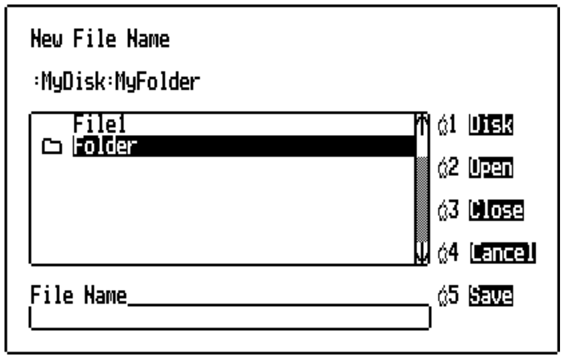
SPACEBAR **Insert Space**

A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that the Insert Space command can extend a line past the end-of-line marker.

CTRLN or ⌘N — New

A dialog like the one show below appears. You need to enter a name for the new file. After entering a name, the editor will open an empty file using one of the ten available file buffers. The file's location on disk will be determined by the directory showing in the dialog's list box.

While the New command requires selecting a file name, no file is actually created until you save the file with the Save command.


New File Name

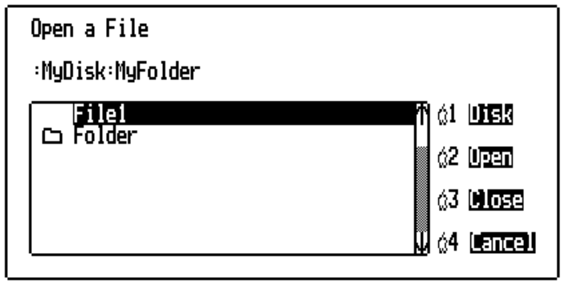
:MyDisk:MyFolder

File1
Folder

File Name

CTRLO or ⌘O — Open

The editor can edit up to ten files at one time. When the open command is used, the editor moves to the first available file buffer, then brings up the dialog shown in Figure 9.4. If there are no empty file buffers, the editor beeps, and the command is aborted.


Open a File

:MyDisk:MyFolder

File1
Folder

Figure 9.4

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the open button, if the selected file in the file list is a TXT or SRC file, the file is opened. If a folder is selected, the folder is opened, and the file list changes to show the files inside the folder. You can also open a file by first selecting a file, then clicking on it with the mouse.

If a folder is open, the close button closes the folder, showing the list of files that contains the folder. You can also close a folder by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

For information on how to use the various controls in the dialog, see "Using Editor Dialogs" in this chapter.

CTRL**V or V** — Paste

The contents of the SYSTEMP file are copied to the current cursor position. If the editor is in line-oriented select mode, the line the cursor is on and all subsequent lines are moved down to make room for the new material. If the editor is in character-oriented select mode, the material is copied at the cursor column. If enough characters are inserted to make the line longer than 255 characters, the excess characters are lost.

CTRL**Q or Q** — Quit

The quit command leaves the editor. If any file has been changed since the last time it was saved to disk, each of the files, in turn, will be made the active file, and the following dialog will appear:

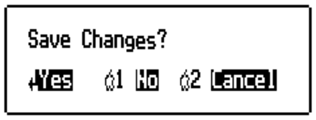

Save Changes?

Figure 9.5

If you select Yes, the file is saved just as if the Save command had been used. If you select No, the file is closed without saving any changes that have been made. Selecting Cancel leaves you in the editor with the active file still open, but if several files had been opened, some of them may have been closed before the Cancel operation took effect.

CTRL**R or R** — Remove Blanks

If the cursor is on a blank line, that line and all subsequent blank lines up to the next non-blank line are removed. If the cursor is not on a blank line, the command is ignored.

1 to 32767 — Repeat Count

When in escape mode, you can enter a *repeat count* (any number from 1 to 32767) immediately before a command, and the command is repeated as many times as you specify (or as many times as is possible, whichever comes first). Escape mode is described in the section “Modes” in this chapter.

RETURN Return

The RETURN key works in one of two ways, depending on the setting of the auto-indent mode toggle: 1) to move the cursor to column one of the next line; or 2) to place the cursor on the first non-space character in the next line, or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor. If the cursor is on the last line on the screen, the screen scrolls down one line.

If the editor is in insert mode, the RETURN key will also split the line at the cursor position.

CTRL**A** or ⌘**A** Save As

The Save As command lets you change the name of the active file, saving it to a new file name or to the same name in a new file folder. When you use this command, this dialog will appear:

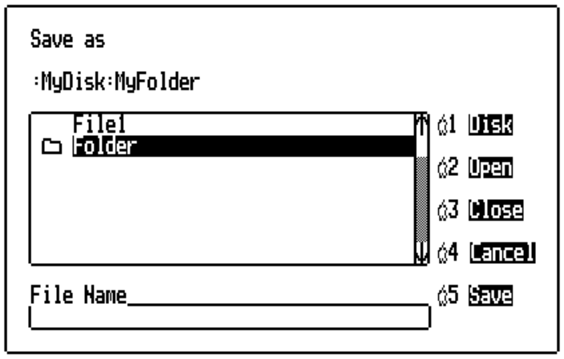


Figure 9.6

Selecting Disk brings up a second dialog that shows a list of the disks available. Selecting one changes the list of files to a list of the files on the selected disk.

When you use the Open button, the selected folder is opened. While using this command, you cannot select any files from the list; only folders can be selected.

If a folder is open, the close button closes the folder, showing the list of files that contains the folder. You can also close a folder by clicking on the path name shown above the file list. If the file list was created from the root volume of a disk, the close button does nothing.

The cancel button leaves the open dialog without opening a file.

The Save button saves the file, using the file name shown in the editline item labeled "File Name." You can also save the file by pressing the RETURN key.

For information on how to use the various controls in the dialog, see "Using Editor Dialogs" in this chapter.

CTRLS or ⌘S — **Save**

The active file (the one you can see) is saved to disk.

⌘-1 to ⌘-9 — **Screen Moves**

The file is divided by the editor into 8 approximately equal sections. The screen-move commands move the file to a boundary between one of these sections. The command ⌘1 jumps to the first character in the file, and ⌘9 jumps to the last character in the file. The other seven ⌘*n* commands cause screen jumps to evenly spaced intermediate points in the file.

⌘} — **Scroll Down One Line**

The editor moves down one line in the file, causing all of the lines on the screen to move up one line. The cursor remains in the same position on the screen. Scrolling can continue past the last line in the file.

⌘] — **Scroll Down One Page**

The screen scrolls down twenty-two lines. Scrolling can continue past the last line in the file.

⌘{ — **Scroll Up One Line**

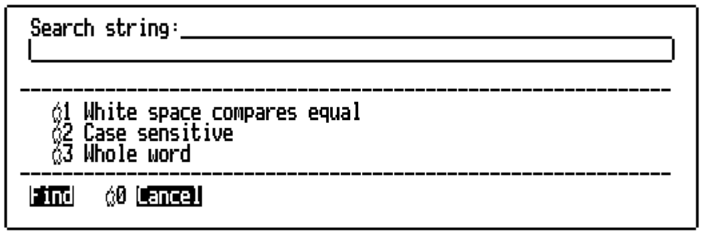
The editor moves up one line in the file, causing all of the lines on the screen to move down one line. The cursor remains in the same position on the screen. If the first line of the file is already displayed on the screen, the command is ignored.

⌘[— **Scroll Up One Page**

The screen scrolls up twenty-two lines. If the top line on the screen is less than one screen's height from the beginning of the file, the screen scrolls to the beginning of the file.

⌘L — **Search Down**

This command allows you to search through a file for a character or string of characters. When you execute this command, the prompt `Search string:` appears at the bottom of the screen.


Search string:

◊1 White space compares equal
◊2 Case sensitive
◊3 Whole word

Find ◊0 Cancel

Figure 9.7

If you have previously entered a search string, the previous string appears after the prompt as a default. Type in the string for which you wish to search, and press RETURN. The cursor will be moved to the first character of the first occurrence of the search string after the old cursor position. If there are no occurrences of the search string between the old cursor position and the end of the file, an alert will show up stating that the string was not found; pressing any key will get rid of the alert.

By default, string searches are case insensitive, must be an exact match in terms of blanks and tabs, and will match any target string in the file, even if it is a subset of a larger word. All of these defaults can be changed, so we will look at what they mean in terms of how changing the defaults effect the way string searches work.

When you look at a line like

```
lb1    lda    #4
```

without using the hidden characters mode, it is impossible to tell if the spaces between the various fields are caused by a series of space characters, two tabs, or perhaps even a space character or two followed by a tab. This is an important distinction, since searching for lda<space><space><space>#4 won't find the line if the lda and #4 are actually separated by a tab character, and searching for lda<tab>#4 won't find the line if the fields are separated by three spaces. If you select the "white space compares equal" option, though, the editor will find any string where lda and #4 are separated by any combination of spaces and tabs, whether you use spaces, tabs, or some combination in the search string you type.

By default, if you search for lda, the editor will also find LDA, since string searches are case insensitive. In C, which is case sensitive, you don't usually want to find MAIN when you type main. Selecting the "case sensitive" option makes the string search case sensitive, so that the capitalization becomes significant. With this option turned on, searching for main would not find MAIN.

Sometimes when you search for a string, you want to find any occurrence of the string, even if it is imbedded in some larger word. For example, if you are scanning your program for places where it handles spaces, you might enter a string like "space". You would want the editor to find the word whitespace, though, and normally it would. If you are trying to scan through a source file looking for all of the places where you used the variable i, though, you don't want the editor to stop four times on the word Mississippi. In that case, you can select the "whole word" option, and the editor will only stop of it finds the letter i, and there is no other letter, number, or underscore character on either side of the letter. These rules match the way languages deal with identifiers, so you can use this option to search for specific variable names – even a short, common one like i.

This command searches from the cursor position towards the end of the file. For a similar command that searches back towards the start of the file, see the "Search Up" command.

For a complete description of how to use the mouse or keyboard to set options and move through the dialog, see the section "Using Editor Dialogs" in this chapter.

Once a search string has been entered, you may want to search for another occurrence of the same string. ORCA ships with two built-in editor macros that can do this with a single keystroke, without bringing up the dialog. To search forward, use the L macro; to search back, use the K macro.

K Search Up

This command operates exactly like Search Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search Down command.

J Search and Replace Down

This command allows you to search through a file for a character or string of characters, and to replace the search string with a replacement string. When you execute this command, the following dialog will appear on the screen:

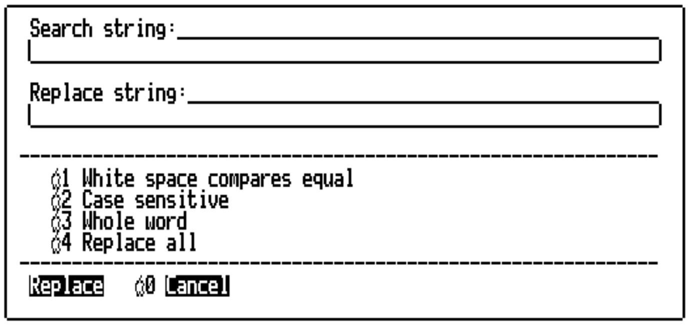
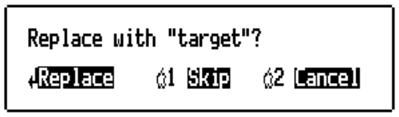


Figure 9.8

The search string, the first three options, and the buttons work just as they do for string searches; for a description of these, see the Search Down command. The replace string is the target string that will replace the search string each time it is found. By default, when you use this command, each time the search string is found in the file you will see this dialog:


Replace with "target"?

↵Replace ⌘1 Skip ⌘2 Cancel

Figure 9.9

If you select the Replace option, the search string is replaced by the replace string, and the editor scans forward for the next occurrence of the search string. Choosing Skip causes the editor to skip ahead to the next occurrence of the search string without replacing the occurrence that is displayed. Cancel stops the search and replace process.

If you use the "replace all" option, the editor starts at the top of the file and replaces each and every occurrence of the search string with the target string. On large files, this can take quite a bit of time. To stop the process, press ⌘. (open-apple period). While the search and replace is going on, you can see a spinner at the bottom right corner of the screen, showing you that the editor is still alive and well.

⌘H Search and Replace Up

This command operates exactly like Search and Replace Down, except that the editor looks for the search string starting at the cursor and proceeding toward the beginning of the file. The search stops at the beginning of the file; to search between the current cursor location and the end of the file, use the Search and Replace Down command. If you use the "replace all" option, this command works exactly the same way the Search and Replace Down command does when it uses the same option.

- Select File

The editor can edit up to ten files at one time. When you use this command, a dialog appears showing the names of the ten files in memory. You can then move to one of the files by pressing n, where n is one of the file numbers. You can exit the dialog without switching files by pressing ESC or RETURN.

See also the Switch Files command.

⌘TAB Set and Clear Tabs

If there is a tab stop in the same column as the cursor, it is cleared; if there is no tab stop in the cursor column, one is set.

[Shift Left

If this command is issued when no text is selected, you enter the text selection mode. Pressing RETURN leaves text selection mode.

At any time while text is selected, using the command shifts all of the selected text left one character. This is done by scanning the text, one line at a time, and removing a space right before the first character on each line that is not a space or tab. If the character to be removed is a tab character, it is first replaced by an equivalent number of spaces. If there are no spaces or tabs at the start of the line, the line is skipped.

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing ., but this will leave the selected text partially shifted.

] Shift Right

If this command is issued when no text is selected, you enter the text selection mode. Pressing RETURN leaves text selection mode.

At any time while text is selected, using the command shifts all of the selected text right one character. This is done by scanning the text, one line at a time, and adding a space right before the first character on each line that is not a space or tab. If this leaves the non-space character on a tab stop, the spaces are collected and replaced with a tab character. If a blank line is encountered, no action is taken.

If a large amount of text is selected, this command may take a lot of time. While the editor is working, you will see a spinner at the bottom right of the screen; this lets you know the editor is still processing text. You can stop the operation by pressing ., but this will leave the selected text partially shifted.

n Switch Files

The editor can edit up to ten files at one time. Each of these files is numbered, starting from 0 and proceeding to 9. The numbers are assigned as the files are opened from the command line. To move from one file to the next, press n, where n is a numeric key.

When you switch files, the original file is not changed in any way. When you return to the file, the cursor and display will be in the same place, the undo buffer will still be active, and so forth. The only actions that are not particular to a specific file buffer are those involving the clipboard – Cut, Copy and Paste all use the same clipboard, so you can move chunks of text from one file to another.

See also the Select File command.

TAB Tab

In insert mode, or when in over strike mode and the next tab stop is past the last character in the line, this command inserts a tab character in the source file and moves to the end of the tab field. If you are in the over strike mode and the next tab stop is not past the last character on the line, the Tab command works like a cursor movement command, moving the cursor forward to the next tab stop.

Some languages and utilities do not work well (or at all) with tab stops. If you are using one of these languages, you can tell the editor to insert spaces instead of tab characters; see the section "Setting Editor Defaults," later in this chapter, to find out how this is done.

TAB — Tab Left

The cursor is moved to the previous tab stop, or to the beginning of the line if there are no more tab stops to the left of the cursor. This command does not enter any characters in the file.

RETURN — Toggle Auto Indent Mode

If the editor is set to put the cursor on column one when you press RETURN, it is changed to put the cursor on the first non-space character; if set to the first non-space character, it is changed to put the cursor on column one. Auto-indent mode is described in the section "Modes" in this chapter.

ESC — Toggle Escape Mode

If the editor is in the edit mode, it is put in escape mode; if it is in escape mode, it is put in edit mode. When you are in escape mode, pressing any character not specifically assigned to an escape-mode command returns you to edit mode. Escape and edit modes are described in the section "Modes" in this chapter.

When in escape mode, CTRL_ will return you to edit mode. In edit mode the command has no effect. From edit mode, CTRL_ will place you in escape mode, but the command has no effect in escape mode. These commands are most useful in an editor macro, where you do not know what mode you are in on entry.

CTRL**E or** **E** — Toggle Insert Mode

If insert mode is active, the editor is changed to over strike mode. If over strike mode is active, the editor is changed to insert mode. Insert and over strike modes are described in the section "Modes" in this chapter.

CTRL**X** — Toggle Select Mode

If the editor is set to select text for the Cut, Copy, and Delete commands in units of one line, it is changed to use individual characters instead; if it is set to character-oriented selects, it is toggled to use whole lines. See the section "Modes" in this chapter for more information on select mode.

UP-ARROW — Top of Screen / Page Up

The cursor moves to the first visible line on the screen, preserving the cursor's horizontal position. If the cursor is already at the top of the screen, the screen scrolls up twenty-two lines. If the cursor is at the top of the screen and less than twenty-two lines from the beginning of the file, then the screen scrolls to the beginning of the file.

CTRLZ or ⌘Z **Undo Delete**

The last operation that changed the text in the current edit file is reversed, leaving the edit file in the previous state. Saving the file empties the undo buffer, so you cannot undo changes made before the last time the file was saved.

The undo operation acts like a stack, so once the last operation is undone, you can undo the one before that, and so on, right back to the point where the file was loaded or the point where the file was saved the last time.

⌘LEFT-ARROW **Word Left**

The cursor is moved to the beginning of the next non-blank sequence of characters to the left of its current position. If there are no more words on the line, the cursor is moved to the last word in the previous line or, if it is blank, to the last word in the first non-blank line preceding the cursor.

⌘RIGHT-ARROW **Word Right**

The cursor is moved to the start of the next non-blank sequence of characters to the right of its current position. If there are no more words on the line, the cursor is moved to the first word in the next non-blank line.

Setting Editor Defaults

When you start the ORCA editor, it reads the file named SYSTABS (located in the ORCA shell prefix), which contains the default settings for tab stops, return mode, insert mode, tab mode, and select mode. The SYSTABS file is an ASCII text file that you can edit with the ORCA editor.

Each language recognized by ORCA is assigned a language number. The SYSTABS file has three lines associated with each language:

1. The language number.
2. The default settings for the various modes.
3. The default tab and end-of-line-mark settings.

The first line of each set of lines in the SYSTABS file specifies the language that the next two lines apply to. ORCA languages can have numbers from 0 to 32767 (decimal). The language number must start in the first column; leading zeros are permitted and are not significant, but leading spaces are not allowed.

The second line of each set of lines in the SYSTABS file sets the defaults for various editor modes, as follows:

1. If the first column contains a zero, pressing RETURN in the editor causes the cursor to go to column one in the next line; if it's a one, pressing RETURN sends the cursor to the first non-space character in the next line (or, if the line is blank, beneath the first non-space character in the first non-blank line on the screen above the cursor).
2. If the second character is zero, the editor is set to line-oriented selects; if one, it is set to character-oriented selects.
3. This flag is not used by the current version of the ORCA editor. It should be set to 0.
4. The fourth character is used by the ORCA/Desktop editor, and is used to set the default cursor mode. A zero will cause the editor to start in over strike mode; a one causes the editor to start in insert mode.
5. If the fifth character is a 1, the editor inserts a tab character in the source file when the Tab command is used to tab to a tab stop. If the character is a 0, the editor inserts an appropriate number of spaces, instead.
6. If the sixth character is a 0, the editor will start in over strike mode; if it is a 1, the editor starts in insert mode. Using a separate flag for the text based editor (this one) and the desktop editor (see the fourth flag) lets you enter one mode in the desktop editor, and a different mode in the text based editor.

The third line of each set of lines in the SYSTABS file sets default tab stops. There are 255 zeros and ones, representing the 255 character positions available on the edit line. The ones indicate the positions of the tab stops. A two in any column of this line sets the end of the line; if the characters extend past this marker, the line is wrapped. The column containing the two then replaces the default end-of-line column (the default right margin) when the editor is set to that language.

For example, the following lines define the defaults for ORCA Assembly Language:

```
8
100100
00000000100000001000000010000000100000001000000010000000000000001000000010000000100000001000000...
```

The last line continues on for a total of 255 characters.

If no defaults are specified for a language (that is, there are no lines in the SYSTABS file for that language), then the editor assumes the following defaults:

- RETURN sends the cursor to column one.
- Line-oriented selects.
- Word wrapping starts in column 80.
- There is a tab stop every eighth column.
- The editor starts in over strike mode.
- Tab characters are inserted to create tabbed text.

Note that you can change tabs and editing modes while in the editor.

Chapter 10 - The Resource Compiler

This chapter describes the use and operation of the resource compiler. Key points covered in this chapter are:

- Creation of resource description files (Rez source files).
- Creating and using resource type statements.
- Using Rez to compile a resource description file to create a resource fork.
- Command, options, and capabilities of the resource compiler.

Overview

The Resource Compiler compiles a text file (or files) called a resource description file and produces a resource file as output. The resource decompiler, DeRez, decompiles an existing resource, producing a new resource description file that can be understood by the resource compiler.

Resource description files have a language type of REZ. By convention, the name of a resource description file ends with .rez. The REZ shell command enables you to set the language type to the rez language.

The resource compiler can combine resources or resource descriptions from a number of files into a single resource file. The resource compiler supports preprocessor directives that allow you to substitute macros, include other files, and use if-then-else constructs. (These are described under "Preprocessor Directives" later in this chapter.)

Resource Decompiler

The DeRez utility creates a textual representation of a resource file based on resource type declarations identical to those used by the resource compiler. (If you don't specify any type declarations, the output of DeRez takes the form of raw data statements.) The output of DeRez is a resource description file that may be used as input to the resource compiler. This file can be edited using the ORCA editor, allowing you to add comments, translate resource data to a foreign language, or specify conditional resource compilation by using the if-then-else structures of the preprocessor.

Type Declaration Files

The resource compiler and DeRez automatically look in the 13:RInclude directory, as well as the current directory, for files that are specified by file name on the command line. They also look in these directories for any files specified by a #include preprocessor directive in the resource description file.

Using the Resource Compiler and DeRez

The resource compiler and DeRez are primarily used to create and modify resource files. The resource compiler can also form an integral part of the process of building a program. For instance, when putting together a desk accessory or driver, you could use the resource compiler to combine the linker's output with other resources, creating an executable program file.

Structure of a Resource Description File

The resource description file consists of resource type declarations (which can be included from another file) followed by resource data for the declared types. Note that the resource compiler and resource decompiler have no built-in resource types. You need to define your own types or include the appropriate .rez files.

A resource description file may contain any number of these statements:

include	Include resources from another file.
read	Read the data fork of a file and include it as a resource.
data	Specify raw data.
type	Type declaration – declare resource type descriptions for subsequent *resource* statements.
resource	Data specification – specify data for a resource type declared in previous *type* statements.

Each of these statements is described in the sections that follow.

A type declaration provides the pattern for any associated resource data specifications by indicating data types, alignment, size and placement of strings, and so on. You can intersperse type declarations and data in the resource description file so long as the declaration for a given resource precedes any resource statements that refer to it. An error is returned if data (that is, a *resource* statement) is given for a type that has not been previously defined. Whether a type was declared in a resource description file or in a #include file, you can redeclare it by providing a new declaration later in a resource description file.

A resource description file can also include comments and preprocessor directives. Comments can be included any place white space is allowed in a resource description file by putting them within the comment delimiters /* and */. Note that comments do not nest. For example, this is one comment:

```
/* Hello /* there */
```

The resource compiler also supports the use of // as a comment delimiter. And characters that follow // are ignored, up to the end of the current line.

```
type 0x8001 { // the rest of this line is ignored
```

Preprocessor directives substitute macro definitions and include files, and provide if-then-else processing before other resource compiling takes place. The syntax of the preprocessor is very similar to that of the C-language preprocessor.

Sample Resource Description File

An easy way to learn about the resource description format is to decompile some existing resources. For example, the following command decompiles only the rIcon resources in an application called Sample, according to the declaration in 13:RInclude:Types.rez.

```
derez sample -only 0x8001 types.rez >derez.out
```

Note that DeRez automatically finds the file types.rez in 13:RInclude. After executing this command, the file derez.out would contain the following decompiled resource:

```
resource 0x8001  (0x1)  {
      0x8000,
      20,
      28
      $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
      $"FFFF FF00 0000 0000 0000 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FF00 FFFF FFFF FF00 FFFF FFFF FFFF"
      $"FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF"
      $"0000 0000 0000 0000 0000 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0FFF FFFF FFFF FFF0 0000 0000 0000"
      $"0000 0000 0000 0000 0000 0000 0000 0000"
};
```

Note that this statement would be identical to the resource description in the original resource description file, with the possible exception of minor differences in formatting. The resource data corresponds to the following type declaration, contained in types.rez:

```
/*------------------------- rIcon -----------------------*/
type rIcon {
        hex integer;                /* Icon Type bit 15  1 = color, 0 = mono */
image:
        integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
        integer;                    /* height of icon in pixels */
        integer;                    /* width of icon in pixels */
        hex string [$$Word(image)]; /* icon image */
mask:
        hex string;                 /* icon mask */
};
```

Type and resource statements are explained in detail in the reference section that follows.

Resource Description Statements

This section describes the syntax and use of the five types of resource description statements available for the resource compiler: include, read, data, type and resource.

Syntax Notation

The syntax notation in this chapter follows the conventions used earlier in the book. In addition, the following conventions are used:

- Words that are part of the resource description language are shown in the Courier font to distinguish them from surrounding text. The resource compiler is not sensitive to the case of these words.

- Punctuation characters such as commas (,), semicolons (;), and quotation marks (' and ") are to be written as shown. If one of the syntax notation characters (for example, [or]) must be written as a literal, it is shown enclosed by "curly" single quotation marks (‘...’); for example,

  `bitstring` ‘[’ *length* ‘]’

  In this case, the brackets would be typed literally – they do *not* mean that the enclosed element is optional.

- Spaces between syntax elements, constants, and punctuation are optional they are shown for readability only.

Tokens in resource description statements may be separated by spaces, tabs, returns, or comments.

There are three terms used in the syntax of the resource description language that have not been used earlier to describe the shell. The are:

resource-ID	A long expression. (Expressions are defined later.)
resource-type	A word expression.
ID-range	A range of *resource-IDs*, as in *ID*[:*ID*].

Include – Include Resources from Another File

The include statement lets you read resources from an existing file and include all or some of them.

An `include` statement can take the following forms:

- `include` *file* [*resource-type* ['`(`'*ID*[:*ID*]'`)`']]`;`

  Read the resource of type *resource-type* with the specified resource ID range in *file*. If the resource ID is omitted, read all resources of the type *resource-type* in *file*. If *resource-type* is omitted, read all the resources in *file*.

- `include` *file* `not` *resource-type* `;`

  Read all resources in *file* that are not of the type *resource-type*.

- `include` *file resource-type1* `as` *resource-type2*`;`

  Read all resources of type *resource-type1* and include them as resources of *resource-type2*.

- `include` *file resource-type1* '`(`'*ID*[:*ID*]'`)`'
  `as` *resource-type2* '`(`'*ID*[`,` *attributes...*]'`)`'`;`

  Read the resource in *file* of type *resource-type1* with the specified ID range, and include it as a resource of *resource-type2* with the specified ID. You can optionally specify resource attributes. (See "Resource Attributes," later in this section.)

Examples:

```
include "otherfile";          /* include all resources from the file */
include "otherfile" rIcon;    /* read only the rIcon resources */
include "otherfile" rIcon (128);    /* read only rIcon resource 128 */
```

AS Resource Description Syntax

The following string variables can be used in the as resource description to modify the resource information in `include` statements:

`$$Type`	Type of resource from include file.
`$$ID`	ID of resource from include file.
`$$Attributes`	Attributes of resource from include file.

For example, to include all `rIcon` resources from one file and keep the same information but also set the preload attribute (64 sets it):

```
INCLUDE "file" rIcon (0:40) AS rIcon ($$ID, $$Attributes | 64);
```

The `$$Type`, `$$ID`, and `$$Attributes` variables are also set and legal within a normal resource statement. At any other time the values of these variables are undefined.

Resource Attributes

You can specify attributes as a numeric expression (as described in the *Apple IIGS Toolbox Reference*, Volume 3) or you can set them individually by specifying one of the keywords from any of the sets in Table 10.1. You can specify more than one attribute by separating the keywords with a comma (,).

Default	Alternative	Meaning
`unlocked`	`locked`	Locked resources cannot be moved by the Memory Manager.
`moveable`	`fixed`	Specifies whether the Memory Manager can move the block when it is unlocked.
`nonconvert`	`convert`	Convert resources require a resource converter.
`handleload`	`absoluteload`	Absolute forces the resource to be loaded at an absolute address.

`nonpurgeable`	`purgeable1`
	`purgeable2`
	`purgeable3`
		Purgeable resources can be automatically purged by the Memory Manager. Purgeable3 are purged before purgeable2, which are purged before purgeable1.

`unprotected`	`protected`	Protected resources cannot be modified by the Resource Manager.
`nonpreload`	`preload`	Preloaded resources are placed in memory as soon as the Resource Manager opens the resource file.
`crossbank`	`nocrossbank`	A crossbank resource can cross memory bank boundaries. Only data, not code, can cross bank boundaries.
`specialmemory`	`nospecialmemory`	A special memory resource can be loaded in banks $00, $01, $E0 and $E1.
`notpagealigned`	`pagealigned`	A page-aligned resource must be loaded with a starting address that is an even multiple of 256.

Table 10.1 Resource Attribute Keywords

Read – Read Data as a Resource

`read` *resource-type* ‘(’ *ID* [, *attributes*] ‘)’ *file* ;

The `read` statement lets you read a file's data fork as a resource. It reads the data fork from *file* and writes it as a resource with the type *resource-type* and the resource ID *ID*, with the optional resource attributes.

Example:

```
read rText (0x1234, Purgeable3) "filename";
```

Data – Specify Raw Data

`data` *resource-type* ‘(’ *ID* [, *attributes*] ‘)’ ‘{’
    data-string
    ‘}’ ;

Use the `data` statement to specify raw data as a sequence of bits, without any formatting.

The data found in *data-string* is read and written as a resource with the type *resource-type* and the ID *ID*. You can specify resource attributes.

When DeRez generates a resource description, it used the data statement to represent any resource type that doesn't have a corresponding type declaration or cannot be decompiled for some other reason.

Example:

```
data rPString (0xABCD) {
   $"03414243"
   };
```

Type – Declare Resource Type

`type` *resource-type* [‘(’ *ID-range* ‘)’] ‘{’
    type-specification...
    ‘}’ ;

A type declaration provides a template that defines the structure of the resource date for a single resource type or for individual resources. If more than one type declaration is given for a resource type the last one read before the data definition is the one that's used. This lets you override declarations from include files of previous resource description files.

After the type declaration, any resource statement for the type *resource-type* uses the declaration `{`*type-specification...*`}`. The optional *ID-range* specification causes the declaration to apply only to a given resource ID or range of IDs.

Type-specification is one or more of the following kinds of type specifier:

```
array          bitstring      boolean        byte           char
cstring        fill           integer        longint        point
pstring        rect           string         switch         wstring
```

You can also declare a resource type that uses another resource's type declaration by using the following variant of the type statement:

`type` *resource-type1* ['`(`' *ID-range* '`)`'] as *resource-type2* ['`(`' *ID* '`)`'] `;`

Integer, Longint, Byte and Bitstring

[`unsigned`] [*radix*] `integer` [= *expression* | *symbol-definition*] `;`
[`unsigned`] [*radix*] `longint` [= *expression* | *symbol-definition*] `;`
[`unsigned`] [*radix*] `byte` [= *expression* | *symbol-definition*] `;`
[`unsigned`] [*radix*] `bitstring` '`[`' *length* '`]`' [= *expression* | *symbol-definition*] `;`

In each case, space is reserved in the resource for an integer or a long integer.

If the type appears alone, with no other parameters, the resource compiler sets aside space for a value that must be given later when the resource type is used to define an actual resource.

A type followed by a equal sign and an expression defines a value that will be preset to some specific integer. Since the value is already given, you do not need to code the value again when the resource type is used to define a resource.

A symbol-definition is an identifier, an equal sign, and an expression, optionally followed by a comma and another symbol definition. It sets up predefined identifier that can be used to fill in the value. You still have the option of coding a numeric value, or you can use one of the constants. This is not a default value, though: you still must code either one of the constants or a numeric value when you use the resource type to define a resource.

The `unsigned` prefix signals DeRez that the number should be displayed without a sign – that the high-order bit can be used for data and the value of the integer cannot be negative. The `unsigned` prefix is ignored by the resource compiler but is needed by DeRez to correctly represent a decompiled number. The resource compiler uses a sign if it is specified in the data. For example, $FFFFFF85 and -$7B are equivalent.

Radix is one of the following constants:

hex decimal octal binary literal

The radix is used by DeRez to decide what number format to use for the output. The radix field is ignored by the resource compiler.

Each of the numeric types generates a different format of integer. In each case, the value is in two's complement form, least significant byte first. The various formats are:

type	size	range
byte	1	-128..255
integer	2	-32768..65535
longint	4	-2147483648..4294967295
bitstring[length]	varies	varies

Sizes are in bytes. The range may seem a little odd at first; the resource compiler accepts either negative or positive values, treating positive values that would normally be too large for a signed value of the given length as if the value were unsigned.

The bitstring type is different from most types in other languages. It is a variable-length integer field, where you specify the number of bits you want as the length field. If you specify a value that only fills part of a byte, then the next field will pick up where the bitstring field stopped. For example, two bitstring[4] values, placed back to back, would require only one byte of storage in the resource file. In general, you should be sure that bitstring fields end on even byte values so the following fields don't get bit aligned to the end of the partially filled byte.

Example:

```
/*---------------------- rToolStartup ---------------------*/
    type rToolStartup {
        integer = 0;                         /* flags must be zero */
        Integer mode320 = 0,mode640 = $80;  /* mode to start quickdraw */
        Integer = 0;
        Longint = 0;
        integer = $$Countof(TOOLRECS);       /* number of tools */
            array TOOLRECS {
                Integer;                     /* ToolNumber */
                Integer;                     /* version */
            };
    };


    resource rToolStartup (1) {
        mode640,
        {
            1,1,      /* Tool Locator */
            2,1,      /* Memory Manager */
            3,1,      /* Miscellaneous Tool Set */
            4,1,      /* QuickDraw II */
            5,1,      /* Desk Manager */
            6,1,      /* Event Manager */
            11,1,     /* Integer Math Tool Set */
            14,1,     /* Window Manager */
            15,1,     /* Menu Manager */
            16,1,     /* Control Manager */
            18,1,     /* QuickDraw II Auxiliary */
            20,1,     /* LineEdit Tool Set */
            21,1,     /* Dialog Manager */
            22,1,     /* Scrap Manager */
            27,1,     /* Font Manager */
            28,1,     /* List Manager */
            30,1,     /* Resource Manager */
        }
    };
```

Boolean

`boolean` [= *constant* | *symbolic-value...*] `;`

A boolean value is a one-bit value, set to either false (0) or true (1). You can also use the numeric values.

True and false are actually predefined constants.

The type boolean is equivalent to

```
unsigned bitstring[1]
```

Example:

```
type 0x001 {
   boolean;
   boolean;
   boolean;
   boolean;
   bitstring[4] = 0;
   };

resource 0x001 (1) {
   true, false, 0, 1
   };
```

Character

`char` [= *string* | *symbolic-value...*] `;`

A character value is an 8-bit value which holds a one-character string. It is equivalent to `string[1]`.

Example:

```
/*----------------------- rMenuItem -----------------------------*/
type rMenuItem {
    integer = 0;                    /* version must be zero */
    integer;                        /* item ID */
    char;                           /* item char */
    char;                           /* alt char */
    integer;                        /* item check */
```

```
        integer;                        /* flags */
        longint;                        /* item titleref */
    };

    resource rMenuItem (1) {
        256,
        "Q","q",
        0,
        0,
        1
        };
```

String, PString, WString and CString

string-type ['`[`' *length* '`]`'] [= *string* | *symbol-value...*] `;`

String types are used to define a string in one of four formats. The format of the string is determined by selecting one of the following for *string-type*:

`[hex] string`	Plain string; no length indicator or terminal character is generated. The optional `hex` prefix tells DeRez to display it as a hexadecimal string. `String[`*n*`]` contains n characters and is *n* bytes long. The type `char` is a shorthand for `string[1]`.
`pstring`	Pascal string; a leading byte containing the number of characters in the string is generated. `Pstring[`*n*`]` contains *n* characters and is *n*+1 bytes long. Since the length must fit in a byte value, the maximum length of a pstring is 255 characters. If the string is too long, a warning is given and the string is truncated.
`wstring`	Word string; this is a very large pstring. The length of a wstring is stored in a two-byte field, giving a maximum length of 65535 characters. `Pstring[`*n*`]` contains *n* characters and is *n*+2 bytes long. The order of the bytes in the length word is least significant byte first; this is the normal order for bytes on the Apple IIGS.
`cstring`	C string; a trailing null byte is added to the end of the characters. `Cstring[`*n*`]` contains *n*-1 characters and is *n* bytes long. A C string of length 1 can be assigned only the value "", since `cstring[1]` only has room for the terminating null.

Each string type can be followed by an optional *length* indicator in brackets. *length* is an expression indicating the string length in bytes. *length* is a positive number in the range 1..2147483647 for string and cstring, in the range 1..255 for pstring, and in the range 1..65535 for wstring.

If no length indicator is given, a pstring, wstring or cstring stores the number of characters in the corresponding data definition. If a length indicator is given, the data may be truncated on the right or padded on the right. The padding characters for all strings are nulls. If the data contains more characters than the length indicator provides for, the string is truncated and a warning message is given.

Examples:

```
/*---------------------- rPString ----------------------*/
type rPString {
        pstring;                    /* String */
};

/*---------------------- rCString ----------------------*/
type rCString {
        cstring;                    /* String */
};

/*---------------------- rWString ----------------------*/
type rWString {
        wstring;                    /* String */
};

/*----------------------- rErrorString ----------------*/
type rErrorString {
        string;
};

resource rPString (1) {
    "p-string",
    };

resource rCString (1) {
    "c-string",
    };

resource rWString (1) {
    "GS/OS input string",
    };

resource rErrorString (1) {
    "Oops",
    };
```

Point and Rectangle

`point` [= *point-constant* | *symbolic-value...*] `;`
`rect` [= *rect-constant* | *symbolic-value...*] `;`

Because points and rectangles appear so frequently in resource files, they have their own simplified syntax. In the syntax shown, a point-constant is defined like this:

‘{’ *x-integer-expression* ‘,’ *y-integer-expression* ‘}’

while a rect-constant looks like this:

‘{’ *integer-expression* ‘,’ *integer-expression* ‘,’ *integer-expression* ‘,’ *integer-expression* ‘}’

A point type creates a pair of integer values, with the first value corresponding to the horizontal point value and the second to the vertical point value. A rect type is a pair of points, with the top left corner of the rectangle specified first, followed by the bottom right corner.

Example:

```
/*-------------------- rWindParam1 -------------------*/
type rWindParam1 {
        integer = $50;              /*length of parameter list, should be $50*/
        integer;                    /* wFrameBits */
        longint;                    /* wTitle */
        longint;                    /* wRefCon */
        rect;                       /* ZoomRect */
        longint;                    /* wColor ID */
        point;                      /* Origin */
        point;                      /* data size */
        point;                      /* max height-width */
        point;                      /* scroll ver hors */
        point;                      /* page vers horiz */
        longint;                    /* winfoRefcon */
        integer;                    /* wInfoHeight */
        fill long[3];               /* wFrameDefProc,wInfoDefProc,wContDefProc
*/
        rect;                       /* wposition */
        longint behind=0,infront=-1;/* wPlane */
        longint;                    /* wStorage */
        integer;                    /* wInVerb */
};

resource rWindParam1 (1) {
        0x80E4,                     /* wFrameBits */
        1,                          /* wTitle */
        0,                          /* wRefCon */
        {0,0,0,0},                  /* ZoomRect */
        0,                          /* wColor ID */
        {0,0},                      /* Origin */
        {416,160},                  /* data size */
        {416,160},                  /* max height-width */
        {0,0},                      /* scroll ver hors */
        {0,0},                      /* page vers horiz */
        0,                          /* winfoRefcon */
        0,                          /* wInfoHeight */
        {32,32,448,192},            /* wposition */
        infront,                    /* wPlane */
        0,                          /* wStorage */
        0x0200                      /* wInVerb */
        };
```

Fill

`fill` *fill-size* [‘`[`’ *length* ‘`[`’] `;`

The resource created by a resource definition has no implicit alignment. It's treated as a bit stream, and integers and strings can start at any bit. The fill specifier is a way of padding fields so that they begin on a boundary that corresponds to the field type.

The fill statement causes the resource compiler to add the specified number of bits to the data stream. The bits added are always set to 0. *fill-size* is one of the following:

```
bit     nibble byte   word    long
```

These declare a fill of 1, 4, 8, 16 or 32 bits, respectively. Any of these can be followed by a *length* modifier. *length* can be any value up to 2147483647; it specifies the number of these bit fields to insert. For example, all of the following are equivalent:

```
fill word[2];
fill long;
fill bit[32];
```

Fill statements are sometimes used as place holders, filling in constant values of zero. You can see an example of the fill statement used for this purpose in the rWindParam1 resource type defined in types.rez. The example in the last section shows this resource type in use.

Array

[`wide`] `array` [*array-name* | ‘`[`’ *length* ‘`]`’] ‘`{`’ *array-list* ‘`}`’ `;`

The *array-list* is a list of type specifications. It can be repeated zero or more times. The `wide` option outputs the array data in a wide display format when the resource is decompiled with DeRez; this causes the elements that make up the *array-list* to be separated by a comma and space instead of a comma, return, and tab.

Either *array-name* or [*length*] may be specified. *Array-name* is an identifier. If the array is named, then a preceding statement should refer to that array in a constant expression with the `$$countof(`*array-name*`)` function, otherwise DeRez will treat the array as an open-ended array. For example,

```
type rToolStartup {
    integer = 0;                         /* flags must be zero */
    Integer mode320 = 0,mode640 = $80;  /* mode to start quickdraw */
    Integer = 0;
    Longint = 0;
```

```
        integer = $$Countof(TOOLRECS);      /* number of tools */
            array TOOLRECS {
                Integer;                    /* ToolNumber */
                Integer;                    /* version */
            };
    };
```

The `$$countof(`*array-name*`)` function returns the number of array elements (in this case, the number of tool number, version pairs) from the resource data.

If length is specified, there must be exactly *length* elements.

Array elements are generated by commas. Commas are element separators. Semicolons are element terminators.

For an example of an rToolStartup resource, see "Integer, Longint, Byte and Bitstream," earlier in this chapter.

Switch

`switch` '{' *case-statement...* '}' `;`

The switch statement lets you select one of a variety of types when you create your resource. Each of the types within the switch statement are placed on a case label, which has this format:

`case` *case-name* `:` *[case-body* `;`] ...

Case-name is an identifier. *Case-body* may contain any number of type specifications and must include a single constant declaration per case, in this form:

`key` *data-type* = *constant*

The key value determines which case applies. For example,

```
/*----------------------- rControlTemplate -----------------------*/
type rControlTemplate {
        integer = 3+$$optionalcount (Fields); /* pCount must be at least 6 */
        longint;                                 /* Application defined ID */
        rect;                               /* controls bounding rectangle */
        switch {

        case SimpleButtonControl:
            key longint = 0x80000000;   /* procRef */
            optional Fields {
                integer;                    /* flags */
                integer;                    /* more flags */
                longint;                    /* refcon */
                longint;                    /* Title Ref */
                longint;                    /* color table ref */
                KeyEquiv;
            };

        case CheckControl:
            key longint = 0x82000000;       /* procRef */
            optional Fields {
                integer;                    /* flags */
                integer;                    /* more flags */
                longint;                    /* refcon */
                longint;                    /* Title Ref */
                integer;                    /* initial value */
                longint;                    /* color table ref */
                KeyEquiv;
            };
        ...and so on.
    };
```

Symbol Definitions

Symbolic names for data type fields simplify the reading and writing of resource definitions. Symbol definitions have the form

name = *value* [, *name* = *value*]…

The "= *value* " part of the statement can be omitted for numeric data. If a sequence of values consists of consecutive numbers, the explicit assignment can be left out; if *value* is omitted, it is assumed to be 1 greater than the previous value. (The value is assumed to be 0 if it is the first value in the list.) This is true for bitstrings (and their derivatives, byte, integer, and longint). For example,

```
integer Emily, Kelly, Taylor, Evan, Trevor, Sparkle=8;
```

In this example, the symbolic names Emily, Kelly, Taylor, Evan, and Trevor are automatically assigned the numeric values 0, 1, 2, 3, and 4.

Memory is the only limit to the number of symbolic values that can be declared for a single field. There is also no limit to the number of names you can assign to a given value; for example,

```
integer    Emily=0, Kelly=1, Taylor=2, Evan=3,
           Trevor=16, Sparkle=0, Twinkle=1, Raphael=2,
           Michaelangelo=3, Nagel=16;
```

Delete – Delete a Resource

`delete` *resource-type* [‘(’ *ID* [: *ID*] ‘)’] ;

This statement deletes the resource of resource-type with the specified ID or ID range from the resource compiler output file. If ID or ID range is omitted, all resources of *resource-type* are deleted.

The delete function is valid only if you specify the –a (append) option on the resource compiler command line. (It wouldn’t make sense to delete a resource while creating a new resource file from scratch.)

You can delete resources that have their protected bit set only if you use the –ov option on the resource compiler command line.

Change – Change a Resource's Vital Information

`change` *resource-type1* [‘(’ *ID* [: *ID*] ‘)’]
    resource-type2 ‘(’ *ID* [, *attributes...*] ‘)’ ;

This statement changes the resource of *resource-type1* with the specified ID or ID range in the resource compiler output file to a resource of *resource-type2* and the specified ID. If ID or ID range is omitted, all resources of *resource-type1* are changed.

The change function is valid only if you specify the –a (append) option on the resource compiler command line. (It wouldn’t make sense to change resources while creating a new resource file from scratch.)

Resource – Specify Resource Data

`resource` *resource-type* ‘(’ *ID* [, *attributes*] ‘)’ ‘{’
    [*data-statement* [, *data-statement*]…]
    ‘}’;

Resource statements specify actual resources, based on previous type declarations.

This statement specifies the data for a resource of type *resource-type* and ID *ID*. The latest type declaration declared for resource-type is used to parse the data specification.

Data statements specify the actual data; data-statements appropriate to each resource type are defined in the next section.

The resource definition generates an actual resource. A resource statement can appear anywhere in the resource description file, or even in a separate file specified on the command line or as an #include file, as long as it comes after the relevant type declaration.

For examples of resource statements, see the examples following the various data statement types, earlier in this chapter.

Data Statements

The body of the data specification contains one data statement for each declaration in the corresponding type declaration. The base type must match the declaration.

Base type	Instance types
string	string, cstring, pstring, wstring, char
bitstring	boolean, byte, integer, longint, bitstring
rect	rect
point	point

Switch data

Switch data statements are specified by using this format:

switch-name data-body

For example, the following could be specified for the rControlTemplate type used in an earlier example:

```
CheckControl { enabled, "Check here" },
```

Array data

Array data statements have this format:

'{' [*array-element* [, *array-element*]…] '}'

where an *array-element* consists of any number of data statements separated by commas.

For example, the following data might be given for the rStringList resource (the type is shown so you won't have to refer to types.rez, where it is defined):

```
type rStringList {
        integer = $$Countof(StringArray);
        array StringArray {
                pstring;            /* String           */
        };
};

resource rStringList (280) {
    {
        "this",
        "is",
        "a",
        "test"
    }
};
```

Sample resource definition

This section describes a sample resource description file for an icon. (See the Apple IIGS Toolbox Reference, Volume 3 for information about resource icons.) The type statement is included for clarity, but would normally be included using an include statement.

```
type rIcon {
        hex integer;            /* icon type bit 15  1 = color,
                                   0 = mono */
image:
        integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
        integer;                    /* height of icon in pixels */
        integer;                    /* width of icon in pixels */
        hex string [$$Word(image)]; /* icon image */
mask:
        hex string;                 /* icon mask */
};

resource rIcon (1) {
        0x8000,                                /* Kind */
        9,                                     /* Height */
        32                                     /* Width */
        $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
        $"FFFFFF00000000000000FFFFFFFFFFFF"
        $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
        $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
        $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
        $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
        $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
        $"FFFFFF00000000000000FFFFFFFFFFFF"
        $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

        $"00000000000000000000000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000FFFFFFFFFFFFFF0000000000000"
        $"00000000000000000000000000000000"
};
```

This data definition declares a resource of type rIcon, using whatever type declaration was previously specified for rIcon. The 8 in the resource type specification (0x8000) identifies this as a color icon.

The icon is 9 pixels high by 32 pixels wide.

The specification of the icon includes a pixel image and a pixel mask.

Labels

Labels support the more complicated resources. Use labels within a resource type declaration to calculate offsets and permit accessing of data at the labels. The rIcon resource, for example, uses labels to specify the pixel image and mask of the icon.

The syntax for a label is:

```
label ::=          character {alphanum}* ':'
character ::=      '_' | A | B | C …
alphanum ::=       character | number
number ::=         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Labeled statements are valid only within a resource type declaration. Labels are local to each type declaration. More than one label can appear on a statement.

Labels may be used in expressions. In expressions, use only the identifier portion of the label (that is, everything up to, but excluding, the colon). See “Declaring Labels Within Arrays” later in this chapter for more information.

The value of a label is always the offset, in bits, between the beginning of the resource and the position where the label occurs when mapped to the resource data. In this example,

```
type 0xCCCC {
    cstring;
endOfString:
    integer = endOfString;
};

resource 0xCCCC (8) {
  "Neato"
}
```

the integer following the cstring would contain:

```
( len("Neato") [5] + null byte [1] ) * 8 [bits per byte] = 48.
```

Built-in Functions to Access Resource Data

In some cases, it is desirable to access the actual resource data to which a label points. Several built-in functions allow access to that data:

- `$$BitField (label, startingPosition, numberOfBits)`

  Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.

- `$$Byte (label)`

  Returns the byte found at *label*.

- `$$Word (label)`

  Returns the word found at *label*.

- `$$Long (label)`

  Returns the long word found at *label*.

For example, the resource type rPString could be redefined without using a pstring. Here is the definition of rPString from Types.rez:

```
type rPString {
    pstring;
};
```

Here is a redefinition of rPString using labels:

```
type rPString {
len:  byte = (stop - len) / 8 - 1;
      string[$$Byte(len)];
stop: ;
};
```

Declaring Labels Within Arrays

Labels declared within arrays may have many values. For every element in the array there is a corresponding value for each label defined within the array. Use array subscripts to access the individual values of these labels. The subscript values range from 1 to n where n is the number of elements in the array. Labels within arrays that are nested in other arrays require multidimensional subscripts. Each level of nesting adds another subscript. The rightmost subscript varies most quickly. Here is an example:

```
type 0xFF01 {
   integer = $$CountOf(array1);
   array array1 {
           integer = $$CountOf(array2);
           array array2 {
foo:                       integer;
           };
   };
};
```

```
resource 0xFF01 (128) {
    {
            {1,2,3},
            {4,5}
    }
};
```

In the example just given, the label foo takes on these values:

```
foo[1,1] = 32        $$Word(foo[1,1]) = 1
foo[1,2] = 48        $$Word(foo[1,2]) = 2
foo[1,3] = 64        $$Word(foo[1,3]) = 3
foo[2,1] = 96        $$Word(foo[2,1]) = 4
foo[2,2] = 112       $$Word(foo[2,2]) = 5
```

Another built-in function may be helpful in using labels within arrays:

`$$ArrayIndex(`*arrayname*`)`

This function returns the current array index of the array *arrayname*. An error occurs if this function is used anywhere outside the scope of the array *arrayname*.

Label Limitations

Keep in mind the fact that the resource compiler and DeRez are basically one-pass compilers. This will help you understand some of the limitations of labels.

To decompile a given type, that type must not contain any expressions with more than one undefined label. An undefined label is a label that occurs lexically after the expression. To define a label, use it in an expression before the label is defined.

This example demonstrates how expressions can have only one undefined label:

```
type 0xFF01 {
    /* In the expression below, start is defined, next is undefined. */
start:      integer = next - start;
    /* In the expression below, next is defined because it was used
       in a previous expression, but final is undefined. */
middle:     integer = final - next;
next:       integer;
final:
};
```

Actually, the resource compiler can compile types that have expressions containing more than one undefined label, but the DeRez cannot decompile those resources and simply generates data resource statements.

The label specified in `$$BitField()`, `$$Byte()`, `$$Word()`, and `$$Long()` must occur lexically before the expression; otherwise, an error is generated.

An Example Using Labels

In the following example, the definition for the rIcon resource uses the labels image and mask.

```
type rIcon {
     hex integer;      /* Icon Type bit 15  1 = color, 0 = mono */
image:
     integer = (Mask-Image)/8 - 6;/* size of icon data in bytes */
     integer;                   /* height of icon in pixels */
     integer;                   /* width of icon in pixels */
     hex string [$$Word(image)]; /* icon image */
mask:
     hex string;                /* icon mask */
};
```

In the data corresponding to that definition, pixel images are provided for the image and mask.

```
resource rIcon (1) {
           0x8000,                     /* Kind */
           9,                                 /* Height */
           32                                 /* Width */
           $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"
           $"FFFFFF00000000000000FFFFFFFFFFFF"
           $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
           $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
           $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
           $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
           $"FFFFFF00FFFFFFFFFF00FFFFFFFFFFFF"
           $"FFFFFF00000000000000FFFFFFFFFFFF"
           $"FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF"

           $"00000000000000000000000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000FFFFFFFFFFFFFF0000000000000"
           $"00000000000000000000000000000000"
};
```

Preprocessor Directives

Preprocessor directives substitute macro definitions and include files and provide if-then-else processing before other resource compiler processing takes place.

The syntax of the preprocessor is very similar to that of the C-language preprocessor. Preprocessor directives must observe these rules and restrictions:

- Each preprocessor statement must begin on a new line, be expressed on a single line, and be terminated by a return character.
- The pound sign (#) must be the first character on the line of a preprocessor statement (except for spaces and tabs).
- Identifiers (used in macro names) may be letters (A–Z, a–z), digits (0–9), or the underscore character (_).
- Identifiers may be any length.
- Identifiers may not start with a digit.
- Identifiers are not case sensitive.

Variable Definitions

The #define and #undef directives let you assign values to identifiers:

`#define` *macro data*
`#undef` *macro*

The #define directive causes any occurrence of the identifier *macro* to be replaced with the text *data*. You can extend a macro over several lines by ending the line with the backslash character (\), which functions as the resource compiler's escape character. Here is an example:

```
#define poem "I wander \
thro\' each \
charter\'d street"
```

Quotation marks within strings must also be escaped. See "Escape Characters: later in this chapter for more information about escape characters.

The #undef directive removes the previously defined identifier macro. Macro definitions can also be removed with the –undef option on the resource compiler command line.

The following predefined macros are provided:

Variable	Value
true	1
false	0
rez	1 or 0 (1 if the resource compiler is running, 0 if DeRez is running)
derez	1 or 0 (0 if the resource compiler is running, 1 if DeRez is running)

If-Then-Else Processing

These directives provide conditional processing:

```
#if expression
[ #elif expression ]
[ #else ]
#endif
```

Expression is defined later in this chapter.
When used with the #if and #elif directives, *expression* may also include one of these terms:

```
defined identifier
defined '(' identifier ')'
```

The following may also be used in place of #if:

```
#ifdef macro
#ifndef macro
```

For example,

```
#define Thai
Resource  rPstring  (199)  {
#ifdef English
   "Hello"
#elif defined (French)
   "Bonjour"
#elif defined (Thai)
   "Sawati"
#elif defined (Japanese)
   "Konnichiwa"
#endif
};
```

Printf Directive

The #printf directive is provided to aid in debugging resource description files. It has the form

```
#printf(formatString, arguments…)
```

The format of the #printf statement is exactly the same as that of the printf statement in the C language, with one exception: There can be no more than 20 arguments. This is the same restriction that applies to the `$$format` function. The #printf directive writes its output to diagnostic output. Note that the #printf directive does not end with a semicolon.

Here’s an example:

```
#define           Tuesday       3
#ifdef Monday
#printf("The day is Monday, day #%d\n", Monday)
#elif defined(Tuesday)
#printf("The day is Tuesday, day #%d\n", Tuesday)
#elif defined(Wednesday)
#printf("The day is Wednesday, day #%d\n", Wednesday)
#elif defined(Thursday)
#printf("The day is Thursday, day #%d\n", Thursday)
#else
#printf("DON'T KNOW WHAT DAY IT IS!\n")
#endif
```

The file just listed generates this text:

```
The day is Tuesday, day #3
```

Formatstring is a text string which is written more or less as is to error out. There are two cases when the string is not written exactly as typed: escape characters and conversion specifiers.

Escape sequences are used to encode characters that would not normally be allowed in a string. The examples show the most commonly used escape sequence, \n. The \ character marks the beginning of an escape sequence, telling the resource compiler that the next character is special. In this case, the next character is n, which indicates a newline character. Printing \n is equivalent to a writeln in Pascal or a PutCR macro from assembly language. For a complete description of escape sequences, see "Escape Characters," later in this chapter.

Conversion Specifiers

Conversion specifiers are special sequences of characters that define how a particular value is to be printed. While the resource compiler actually accepts all of the conversion specifiers allowed by the C language (it is written in C, and uses C's sprintf function to format the string for this statement), many of the conversion specifiers that are used by C are not useful in the resource compiler, and some of the others are not commonly used. For example, technically the resource compiler supports floating-point output, but it does not have a floating point variable type, so the conversion specifiers for floating point values are not of much use. Only those conversion specifiers that are generally used in the resource compiler will be covered here.

Each conversion specifier starts with a % character; to write a % character, code it twice, like this:

```
printf("100%%\n");
```

Conversion specifiers are generally used to write string or numeric arguments. For example, the %n conversion specifier is used to write a two-byte integer. You can put one of several characters between the % characters that starts a conversion specifier and the letter character that indicates the type of the argument; each of these additional characters modifies the format specifier in some way. The complete syntax for a format specifier is

`%` *flag* [*field-width*] [*size-specifier*] *conversion*

Flag is one or more of the characters -, 0, + or a space. The entire field is optional. These flags effect the way the output is formatted:

-	If a formatted value is shorter than the minimum field width, it is normally right-justified in the field by adding characters to the left of the formatted value. If the - flag is used, the value is left-justified.
0	If a formatted value is shorter than the minimum field width, it is normally padded with space characters. If the 0 flag is used, the field is padded with zeros instead of spaces. The 0 pad character is not used if the value is left-justified.
+	Forces signed output, adding a + character before positive integers.
space	Adds a space before positive numbers (instead of a +) so they line up with collimated negative numbers.

Field-width gives the number of characters to use for the output field. If the number of characters needed to represent a value is less than the field width, spaces are added on the left to fill out the field. For example, the statement

```
printf("%10n%10n\n", a, b);
```

could be used to print two columns of numbers, where each column is ten characters wide and the numbers are right-justified.

The *size-specifier* gives the size of the operand. If the *size-specifier* is omitted, the resource compiler expects to find an integer parameter in the parameter list when it processes any of the numeric conversion specifiers. If the size specifier is h, a byte is expected, while l indicates that the resource compiler should look for a longint value.

Conversion tells what size and type of operand to expect and how to format the operand:

Conversion	Format
`d`	signed integer
`u`	unsigned integer
`o`	unsigned octal integer
`x`	unsigned hexadecimal number; lowercase letters are used
`X`	unsigned hexadecimal number; uppercase letters are used
`c`	character
`s`	c-string
`p`	p-string
`%`	write a single % character

You must include exactly one parameter after the format string for each conversion specifier in the format string, and the types of the parameters must agree exactly with the types indicated by the conversion specifiers. Parameters are matched with conversion specifiers on a left-to-right basis.

Include Directive

The #include directive reads a text file:

`#include "`*filename*`"`

The directive behaves as if all of the lines in *file* were placed in the current source file, replacing the line with the directive. The maximum nesting is to ten levels. For example,

```
#include ($$Shell("ORCA")) "MyProject MyTypes.rez"
```

Note that the #include preprocessor directive (which includes a file) is different from the previously described include statement, which copies resources from another file.

The #include directive will look up to three places for the file, in order:

1. The current directory.
2. The directory where the source file is located (generally the current directory, but not always).
3. The directory 13:RInclude.

Append Directive

This directive allows you to specify additional files to be compiled by the resource compiler. The format is:

`#append "`*filename*`"`

This directive must appear between resource or type statements. The *filename* variable is the name of the next file to be compiled. The same search rules apply here that apply to the #include directive. Normally you should place this directive at the end of a file because everything after it is ignored. Do not place a #append directive in an include file.

If you use more than one #append directive, the order in which you put them is important. When the resource compiler sees an #append directive, it checks the language type of the appended file. If it is the same language, that is, REZ, the effect is the same as if the files had been concatenated into a single file. If they are in different languages, the shell quits the resource compiler and begins a new assembly or compilation. Two examples will illustrate why the order is important.

In the first example, suppose you have the following three files, each appended to the preceding file.

```
file1.rez
file2.rez
file3.asm
```

The Compile command calls the resource compiler to process file1.rez because the language is REZ. When the resource compiler encounters the #append directive for file2.rez it continues processing as if file.rez and file2.rez had been concatenated into a single file. When it encounters the #append directive for file3.asm, the resource compiler finishes processing and returns control to the shell which calls the assembler to assemble file3.asm.

The result is different if the order of the files is changed, as follows:

```
file1.rez
file3.asm
file2.rez
```

The resource compiler processes file1.rez. When it encounters the #append directive for file3.asm, the resource compiler finishes processing and returns control to the ORCA shell because the language stamp is different. The shell calls the assembler to processes file3.asm. When the assembler is finished processing, it returns control to the shell which calls the resource compiler to process file2.rez. However, since this is a separate compilation from that of file1.rez, the resource compiler knows nothing about symbols from file1.rez when compiling file2.rez.

DeRez handles #append directives differently from the resource compiler. For DeRez the file being appended must have a language stamp of REZ or DeRez will treat the #append directive as an end-of-file marker. DeRez will not return control to the shell after finishing processing. Therefore, in the previous example, DeRez would process file1.rez only and then finish processing.

Resource Description Syntax

This section describes the details of the resource description syntax.

Numbers and Literals

All arithmetic is performed as 32-bit signed arithmetic. The basic formats are shown in Table 10.2.

Numeric Type	Form	Meaning
Decimal	nnn…	Signed decimal constant between 2,147,483,647 and –2,147,483,648. Do not use a leading zero. (See octal.)
Hexadecimal	0Xhhh…	Signed hexadecimal constant between 0X7FFFFFFF and 0X80000000.

	$hhh…	Alternate form for hexadecimal constants.
Octal	0ooo…	Signed octal constant between 017777777777 and 020000000000. A leading zero indicates that the number is octal.
Binary	0Bbbb…	Signed binary constant between 0B11111111111111111111111111111111 and 0B10000000000000000000000000000000.
Literal	'aaaa'	One to four printable ASCII characters or escape characters. If there are fewer than four characters in the literal, the characters to the left (high bits) are assumed to be $00. Characters that are not in the printable character set, and are not the characters \' and \\ (which have special meanings), can be escaped according to the character escape rules. (See “Strings” later in this section.)

Table 10.2: Numeric Constants

Literals and numbers are treated in the same way by the resource compiler. A literal is a value within single quotation marks; for instance, 'A' is a number with the value 65; on the other hand, "A" is the character A expressed as a string. Both are represented in memory by the bitstring 01000001. (Note, however, that "A" is not a valid number and 'A' is not a valid string.) The following numeric expressions are all equivalent:

```
'B'            66             'A'+1
```

Literals are padded with nulls on the left side so that the literal 'ABC' is stored as shown in Figure 10.3.

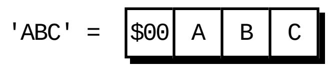


Figure 10.3: Padding of Literals

Expressions

An expression may consist of simply a number or a literal. Expressions may also include numeric variables, labels, and system functions.

Table 10.3 lists the operators in order of precedence with highest precedence first – groupings indicate equal precedence. Evaluation is always left to right when the priority is the same.

Precedence	Operator	Meaning
1.	(expr)	Forced precedence in expression calculation
2.	-expr	Arithmetic (two’s complement) negation of expr
	~expr	Bitwise (one’s complement) negation of expr
	!expr	Logical negation of expr

3.	expr1 * expr2	Multiplication
	expr1 / expr2	Integer division
	expr1 % expr2	Remainder from dividing expr1 by expr2
4.	expr1 + expr2	Addition
	expr1 - expr2	Subtraction
5.	expr1 << expr2	Shift left; shift expr1 left by expr2 bits
	expr1 >> expr2	Shift right; shift expr1 right by expr2 bits
6.	expr1 > expr2	Greater than
	expr1 >= expr2	Greater than or equal to
	expr1 < expr2	Less than
	expr1 <= expr2	Less than or equal to
7.	expr1 == expr2	Equal
	expr1 != expr2	Not equal
8.	expr1 & expr2	Bitwise AND
9.	expr1 ^ expr2	Bitwise XOR
10.	expr1 | expr2	Bitwise OR
11.	expr1 && expr2	Logical AND
12.	expr1 || expr2	Logical OR

Table 10.3: Resource Description Operators

The logical operators !, >, >=, <, <=, ==, !=, &&, and || evaluate to 1 (true) or 0 (false).

Variables and Functions

There are several predefined variables that are preset by the resource compiler, or that take on specific meaning based on how they are used in your resource description file. Some of these resource compiler variables also contain commonly used values. All Rez variables start with $$ followed by an alphanumeric identifier.

The following variables and functions have string values:

`$$Date` Current date. It is useful for putting time-stamps into the resource file. The format of the string is: weekday, month dd, yyyy. For example, August 10, 1989.

`$$Format("`*formatString*`",` *arguments*`)`
Works just like the #printf directive except that `$$Format` returns a string rather than printing to standard output. (See “Print Directive” earlier in this chapter.)

`$$Resource("`*filename*`",'`*type*`',`*ID*`)`
Reads the resource `'`*type*`'` with the ID *ID* from the resource file *filename*, and returns a string.

`$$Shell("`*stringExpr*`")`	Current value of the exported shell variable {stringExpr }. Note that the braces must be omitted, and the double quotation marks must be present.
`$$Time`	Current time. It is useful for time-stamping the resource file. The format is: "hh:mm:ss".
`$$Version`	Version number of the resource compiler. (“V1.0”)

These variables and functions have numeric values:

`$$Attributes`	Attributes of resource from the current resource.
`$$BitField(`*label*`,` *startingPosition*`,` *numberOfBits*`)`	Returns the *numberOfBits* (maximum of 32) bitstring found *startingPosition* bits from *label*.
`$$Byte(`*label*`)`	Returns the byte found at *label*.
`$$CountOf (`*arrayName*`)`	Returns the number of elements in the array *arrayName*.
`$$Day`	Current day (range 1–31).
`$$Hour`	Current hour (range 0–23).
`$$ID`	ID of resource from the current resource.
`$$Long(`*label*`)`	Returns the long word found at *label*.
`$$Minute`	Current minute (range 0–59).
`$$Month`	Current month (range 1–12).
`$$OptionalCount (`*OptionalName*`)`	Returns the number of items explicitly specified in the block *OptionalName*.
`$$PackedSize(`*Start*`,` *RowBytes*`,` *RowCount*`)`	Given an offset (*Start*) into the current resource and two integers, *RowBytes* and *RowCount*, this function calls the toolbox routine UnpackBytes *RowCount* times. `$$PackedSize( )` returns the unpacked size of the data found at *Start*. Use this function only for decompiling resource files. An example of this function is found in Pict.rez.

`$$ResourceSize`	Current size of resource in bytes. When decompiling, `$$ResourceSize` is the actual size of the resource being decompiled. When compiling, `$$ResourceSize` returns the number of bytes that have been compiled so far for the current resource.
`$$Second`	Current second (range 0–59).
`$$Type`	Type of resource from the current resource.
`$$Weekday`	Current day of the week (range 1–7, that is, Sunday–Saturday).
`$$Word(`*label*`)`	Returns the word found at *label*.
`$$Year`	Current year.

Strings

There are two basic types of strings:

Text string	`"a…"`	The string can contain any printable character except ‘ " ’ and ‘\’. These and other characters can be created through escape sequences. (See Table 10-4.) The string "" is a valid string of length 0.
Hexadecimal string	`$"hh…"`	Spaces and tabs inside a hexadecimal string are ignored. There must be an even number of hexadecimal digits. The string $"" is a valid hexadecimal string of length 0.

Any two strings (hexadecimal or text) will be concatenated if they are placed next to each other with only white space in between. (In this case, returns and comments are considered white space.)

Figure 10.4 shows a p-string declared as

```
pstring [10];
```

whose data definition is

```
"Hello"
```

$05	H	e	l	l	o	$00	$00	$00	$00	$00

Figure 10.4: Internal Representation of a P-string

In the input file, string data is surrounded by double quotation marks ("). You can continue a string on the next line. A separating token (for example, a comma) or brace signifies the end of the string data. A side effect of string continuation is that a sequence of two quotation marks ("") is simply ignored. For example,

```
"Hello ""out "
"there."
```

is the same string as

```
"Hello out there.";
```

To place a quotation mark character within a string, precede the quotation mark with a backslash, like this:

```
\"
```

Escape Characters

The backslash character (\) is provided as an escape character to allow you to insert nonprintable characters in a string. For example, to include a newline character in a string, use the escape sequence

```
\n
```

Valid escape sequences are shown in Table 10.4.

Escape Sequence	Name	Hexadecimal Value	Printable Equivalent
\t	Tab	$09	None
\b	Backspace	$08	None
\r	Return	$0A	None
\n	Newline	$0D	None
\f	Form feed	$0C	None
\v	Vertical tab	$0B	None
\?	Rub out	$7F	None
\\	Backslash	$5C	\
\'	Single quotation mark	$27	'
\"	Double quotation mark	$22	"

Table 10.4: Resource Compiler Escape Sequences

Note to C programmers: The escape sequence \n produces an ASCII code of 13 in the output stream, while the \r sequence produces an ASCII code of 10. This is backwards from the way the

C language uses these two characters, so if you are creating string resources that will be used with stdio functions from the standard C library, be sure and use \r in your resource file any time you would use \n in C, and use \n in your resource file any time you would use \r in C.

You can also use octal escape sequences, hexadecimal escape sequences, decimal escape sequences and binary escape sequences to specify characters that do not have predefined escape equivalents. The forms are:

Base	Number Form	Digits	Example
2	\0Bbbbbbbbb	8	\0B01000001
8	\ooo	3	\101
10	\0Dddd	3	\0D065
16	\0Xhh	2	\0X41
16	\$hh	2	\$41

Since escape sequences are imbedded in strings, and since these sequences can contain more than one character after the \ character, the number of digits given for each form is an important consideration. You must always code exactly the number of digits shown, using leading zeros if necessary. For example, instead of `"\0x4"`, which only shows a single hexadecimal digit, you must use `"0x04"`. This rule avoids confusion between the numeric escape sequence and any characters that might follow it in the string.

Here are some examples:

```
\077                    /* 3 octal digits */
\0xFF                   /* ‘0x’ plus 2 hex digits */
\$F1\$F2\$F3            /* ‘$’ plus 2 hex digits */
\0d099                  /* ‘0d’ plus 3 decimal digits */
```

You can use the DeRez command-line option –e to print characters that would otherwise be escaped (characters preceded by a backslash, for example). Normally, only characters with values between $20 and $7E are printed as Apple IIGS characters. With this option, however, all characters (except null, newline, tab, backspace, form-feed, vertical tab, and rub out) will be printed as characters, not as escape sequences.

Using the Resource Compiler

The Resource Compiler is a one-pass compiler; that is, in one pass it resolves preprocessor macros, scans the resource description file, and generates code into a code buffer. It then writes the code to a resource file.

The resource compiler is invoked by the shell's compile (or assemble) command, just as you would assemble a program. This command checks the language type of the source file (in this case, rez) and calls the appropriate compiler or assembler (in this case, the resource compiler). In short, with the exception of a few resource compiler specific options, you use the same commands to create a resource fork from a resource description file that you would use to assemble a program.

Resource Forks and Data Forks

Files on the Apple IIGS actually have two distinct parts, known as the data fork and the resource fork. The data fork is what is traditionally a file on other computers; this is where the executable program is stored, where ASCII text is placed for a text file, and so forth. When the resource compiler writes resources, it writes them to the resource fork of the file. Writing to the resource fork of an existing file does not change the data fork in any way, and writing to the data fork does not change the resource fork. The implications of this can speed up the development cycle for your programs. When you compile a resource description file to create a resource fork for your program, you can and should have the resource compiler save the resource fork to the same file in which the linker places the executable code. When you make a change to your assembly language source code, you will normally assemble and link the changed program, creating an updated data fork for your program. If the resource description file has not changed, you do not need to recompile the resource description file. The same is true in reverse: if you make a change to the resource description file, you need to recompile it, but you do not need to reassemble or relink your assembly language source file.

Rez Options

The resource compiler supports the e, s, and t flags from the assemble or compile command. It ignores all other flags.

The resource compiler supports a number of language dependent options. These are coded as the name of the language, an equal sign, and the option list, enclosed in parenthesis. Like the other parameters for the compile command, no spaces are allowed outside of the parenthesis.

For example, the following compile command uses the options list to specify the -p flag, which turns on progress information.

```
compile resources keep=program rez=(-p)
```

The resource compiler will accept up to 31 options in the options list. Any others are ignored.

Here's a complete list of the options that can be used in this options field:

`–a[ppend]` This option appends the resource compiler’s output to the output file's resource fork, rather than replacing the output file's resource fork.

`–d[efine]` *macro* [=*data*]

This option defines the macro variable *macro* to have the value *data*. If data is omitted, macro is set to the null string – note that this still means that macro is defined. Using the –d option is the same as writing

```
#define macro [ data ]
```

at the beginning of the input.

`–flag SYSTEM`	This option sets the resource file flag for the system.
`–flag ROM`	This option sets the resource file flag for ROM.
`–i` *pathname(s)*	This option searches the following path names for #include files. It can be specified more than once. The paths are searched in the order they appear on the command line. For example,

```
…rez=(–i 13:rinclude:stuff.rez
      –i 13:rinclude:newstuff.rez)
```

`–m[odification]`	Don't change the output file's modification date. If an error occurs, the output file's modification date is set, even if you use this option.
`–ov`	This option overrides the protected bit when replacing resources with the –a option.
`–p[rogress]`	This option writes version and progress information to diagnostic output.
`–rd`	This option suppresses warning messages if a resource type is redeclared.
`–s` *pathname(s)*	This option searches the following path names for resource include files.
`–t[ype]` *typeExpr*	This option sets the type of the output load file to *filetype*. You can specify a hexadecimal number, a decimal number, or a mnemonic for the file type. If the –t option is not specified, the file type of the load file is $B3.
`–u[ndef]` *macro*	This option undefines the macro variable *macro*. It is the same as writing

```
#undef macro
```

at the beginning of the input. It is meaningful to undefine only the preset macro variables.

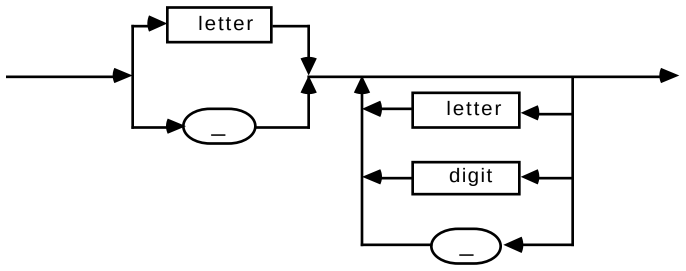
Note: A space is required between an option and its parameters.

Chapter 11 - Program Symbols

Pascal programs are made up of a series of program symbols called tokens. Tokens are the words used to write a program. They consist of identifiers, symbols, and constants.

Identifiers

identifier



Identifiers are the names that you create to represent variables, types, procedures, and so on. Identifiers always begin with an alphabetic character or underscore. They are followed by zero or more alphabetic characters, numeric characters, and underscore characters. Pascal is a case-insensitive language, which means that the identifiers `Name` and `name` represent the same item. The underscore character is significant, however, so that `name` and `name_` are not the same identifiers.

ISO The use of the underscore character in ORCA/Pascal is an extension to Standard Pascal. If portability is an issue, avoid the use of the underscore character. ∆

Identifiers can be any length in Pascal, and all characters are significant. ORCA/Pascal limits the length of a source line to 255 characters. Since identifiers must appear on a single source line, this gives an effective limit of 255 characters for a single identifier.

Reserved Words

Forty-four identifiers are reserved in ORCA/Pascal. They can only be used in the context specified by the Pascal language. They can never be redefined. A reserved word can be used as a part of another identifier, so long as it is not used alone. For example, `myprocedure` is a legal Pascal identifier.

The reserved words are:

```
and            end              inherited     packed        type
array          file             label         procedure     unit
begin          for              mod           program       univ
case           function         nil           record        until
const          goto             not           repeat        uses
div            if               object        set           var
do             implementation   of            string        while
downto         in               or            then          with
else           interface        otherwise     to
```

ISO

The reserved words `object`, `inherited`, `string`, `uses`, `interface`, `implementation`, `unit`, `univ`, and `otherwise` are extensions to Standard Pascal. Avoid their use if portability is an issue. Δ

Reserved Symbols

The reserved symbols are the punctuation marks and mathematical symbols used in Pascal. Each reserved symbol must be typed without intervening spaces. The reserved symbols are:

```
+    -    *    /    =    <    >    !    ~
[    ]    .    ,    :    ;    ^    <<   |
(    )    <>   <=   >=   :=   ..   >>   &
```

The caret character ^ will appear as an up-arrow in some type faces. The up-arrow and caret character are equivalent. Pascal also allows the use of alternate symbols for three of the reserved symbols. The reserved symbols that have alternate forms, and their alternates, are:

Symbol	Alternate
^	@
[	(.
]	.)

The alternate reserved symbols can be exchanged for the normal reserved symbol without changing the meaning of the program. Both the alternate and normal forms can be mixed in the same program.

ISO

The @ character can only be used as a substitute for the ^ character if the ISO+ directive has been declared in your program. If it has not, the @ character is a reserved symbol used to extract the address of a structure. Δ

ISO

The reserved symbols <<, >>, ~, |, ! and & are extensions to Standard Pascal. Avoid their use if portability is an issue. Δ

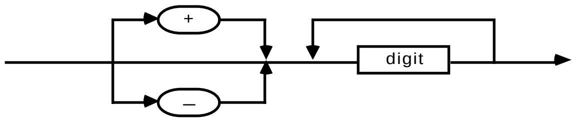
Comments are not normally considered reserved symbols, nor are the characters that delimit comments. See the section on separators, below, for a discussion of comment characters.

Constants

The last class of token is the constant. There are five kinds of constants that can be entered in a Pascal program: integers, hexadecimal numbers, real numbers, characters, and strings.

Integers

signed integer

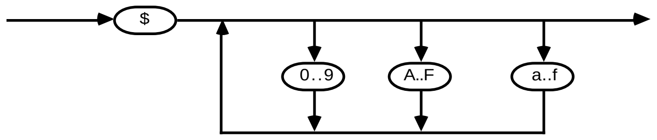


Integers are made up of a series of digits. They come in two sizes: two-byte integers and four-byte integers. A complete discussion of integers and their sizes can be found in Chapter 12. The important point here is that two-byte integers must be in the range -32767..32767, while four-byte integers can be in the range -2147483647..2147483647. Four-byte integers are implemented as an extension to Standard Pascal. They are not valid if the ISO+ directive has been used to force conformance with Standard Pascal. In that case, integer constants outside of the range -32767..32767 will be flagged as an error. Some legal integer constants are:

```
1          32767       99               -100000
```

Hexadecimal Numbers

hexadecimal digit



ISO Hexadecimal constants are an extension to Standard Pascal. Avoid their use if portability is an issue. Δ

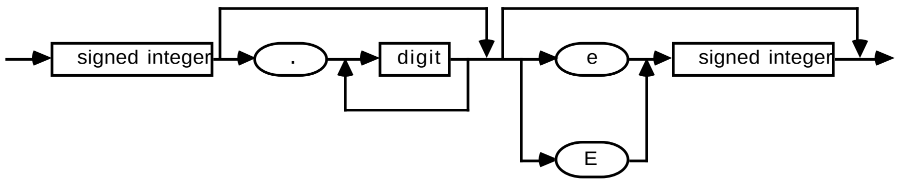
Hexadecimal numbers can be used anywhere an integer constant would be used. They are coded by beginning them with a dollar sign ($), and following with one or more hexadecimal digits

(0 - 9, a - f, A - F). The constant is stored as an integer if there are four or fewer hexadecimal digits, or as a longint if are five or more digits.

There are two ways to interpret a hexadecimal value in the range $8000..$FFFF. If $FFFF is interpreted as a two-byte integer, it's value is -1. Interpreting the same number as a four-byte value, the value would be 65535. Since this hexadecimal constant is coded with four digits, the first meaning is used. To code 65535 as a four-byte, positive hexadecimal value, write the constant as $0FFFF.

Real Numbers

signed real



Real numbers consist of an integer followed by a decimal point, a fraction part, an `e` or `E`, and a signed integer. The decimal point and fraction part can be left out if the exponent is coded, or the exponent can be left out if there is a fractional part. One of the two must appear to distinguish the real constant from an integer.

These rules mean that the following real constants are not legal.

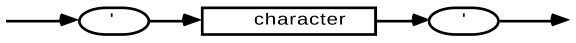
```
1.                      {fraction part is missing}
.3                      {integer part is missing}
8.4e                    {missing exponent}
```

Some legal real constants are:

```
3276.8e10               1.0                     0.3
3.14159                 1e-10                   14.5E+16
```

Character Constants

character constant



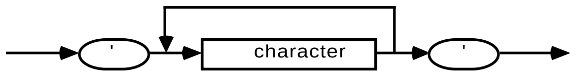
Character constants consist of any keyboard character surrounded by single quote marks. If the character is a single quote mark, double it. The following are all legal character constants:

Character constant	Character value
`' '`	`< blank >`
`''''`	`'`
`';'`	`;`
`'A'`	`A`
`'a'`	`a`

Note that, although the Pascal language is case insensitive, character constants are not. `'A'` is not the same as `'a'`.

String Constants

string constant



ISO Standard Pascal does not allow the use of a null string (a string with no characters). ORCA Pascal allows the null string. Δ

Strings are keyboard characters enclosed in quote marks. Like character constants, quote marks in strings are doubled, and strings are case sensitive. The string and its surrounding quote marks must appear on a single source line, which effectively limits the number of characters in a single string to 253. Character constants and strings with one character are distinguishable by the context in which they are used. The following are legal strings in Pascal:

```
'Now''s the time to program.'
'   '
'!@#$%^&*()'
```

For a discussion on the internal format of a string constant, see "Strings" in the next chapter.

Separators

Separators consist of blanks, the end of a line, and comments. Separators can be used between any two tokens, or before the first token of a program. All separators are completely interchangeable. You can type a Pascal program on one line (if it is short enough), or place every token on a new line. The program will execute the same way.

Separators must be used between any two adjacent labels, numbers, reserved words or identifiers.

Comments begin with a `{` character and end with a `}` character. The characters between the comment characters can be any keyboard characters except `}` or `*)`. Comments are for your convenience only – replacing a comment with a space will have no effect on the finished program. Note that nested comments are not allowed. `{{...}}` is not a legal comment in Pascal.

Both for compatibility with older versions of Pascal and for use on machines that may not have the `{` and `}` characters, `(*` can be substituted for `{`, and `*)` can be substituted for `}`. The two forms of comment symbols can be freely intermixed.

Some legal comments are:

```
{A simple comment}
(*The alternate form*)
{The two can be mixed*)
(*in either direction}
{*)
(*}
{{{{{{{{{{}
{-----------}
{}
(**)
```

Unlike any token in the Pascal language, comments can be spread over more than one source line.

Chapter 12 - Basic Data Types

Pascal has a rich variety of data types, many of which are defined when a program is written. This chapter describes those Pascal data types which are built into the language. The next chapter covers derived and defined data types. Chapter 14 covers objects, which are part data and part procedure and function declaration.

Some of the information in this chapter deals with the way that information is stored internally in the program. This information is provided for very advanced programmers who need to write assembly language subroutines that will deal with Pascal data, or who need to do strange and dangerous tricks with the data to work with the machine at the hardware level. You do not need to understand this information to use ORCA/Pascal for normal Pascal programming. If it does not make sense to you, or if you will not be using the information, simply ignore it.

Integers

Integers are whole numbers. Valid values for integers range from -32767 to 32767. Each integer variable requires two bytes of storage. In this implementation of ORCA/Pascal, integers are stored in two's complement form with the least significant byte first.

ISO The types longint and byte are not a part of Standard Pascal. ∆

ORCA/Pascal supports an extended integer data type, called longint. Longint values require four bytes of storage. The can range from -2147483647 to 2147483647. Longint values can generally be used anywhere that an integer value is allowed. A variable of type longint *cannot* be used as a subscript, as the incrementing variable in a FOR statement, or in a set. When longint values appear in expressions with integer values, the two-byte integer is first promoted to a four-byte value, then the operation is performed.

ORCA/Pascal also supports a predefined subrange of integer, named byte. Values of type byte range from 0 to 255. While byte requires only eight bits of storage, you should use the type sparingly, since the code generated by the compiler to access and manipulate a single byte of memory on the Apple IIGS outweighs any savings in memory gained from using this type of data. We suggest that the byte type be limited to large arrays, files, and toolbox parameters.

Reals

Real numbers are a limited precision, limited range subset of real numbers from mathematics. Real numbers range in absolute value from 1.2e-38 to 3.4e+38. They can, of course, have either a positive or a negative sign. Real numbers are accurate to seven significant figures.

Internally, real numbers are represented using the format specified by the IEEE floating-point standard. Each real number requires four bytes of storage. The exact format will not be specified here, since it has no real bearing on programs written entirely in Pascal - for details, see the IEEE floating-point standard, the SANE reference manual, or the ORCA/M assembly language reference manual.

ISO The types double, extended and comp are not a part of Standard Pascal. Δ

ORCA/Pascal provides three extensions to real numbers. The first is double, which is represented internally using the IEEE floating-point format. Each double-precision number requires eight bytes of storage. Double-precision numbers range in absolute value from 2.3e-308 to 1.7e+308, and can be either positive or negative. Double values are accurate to fifteen significant digits.

The second is extended, which is represented internally using an extended form of the IEEE floating-point format. Each extended-precision number requires ten bytes of storage. Extended-precision numbers range in absolute value from 1.7e-4932 to 1.1e+4932, and can be either positive or negative. Extended values are accurate to nineteen significant digits.

The extended format is the format used internally by Apple's SANE floating point routines, which are called by ORCA/Pascal. In some cases, a program will actually run faster if the numbers are extended than if they are real, since the number doesn't need to be converted back and forth between the internal formats.

The last of the real types is comp, which is represented internally as an eight-byte integer. Comp numbers range from about -9.2e18 to 9.2e18.

The four types or real numbers are completely interchangeable. All intermediate results are manipulated internally in extended format.

Sets

A set is essentially a list of the items from a given base type that are contained in a given set variable. For example, a set can be defined as a `set of 1..10`, in which case the set variable can hold integers in [1..10]. Sets can then be manipulated using a special group of set operations.

The base type of a set must be a scalar. Sets in ORCA/Pascal can hold up to 2048 elements. The apparent range of the scalar variables must be in [0..2047].

Internally, sets are actually variable length. When the set is declared, the largest value that can be an element of the set determines the size of the set variable. Eight set elements are held in each byte, counting from zero, so if `large` is the value of the largest set element, then the set is

```
large mod 8 + 1
```

bytes long. Within each byte, the sets are assigned to bits counting from the least significant bit position, so that set element number 0 will be the least significant bit of the first byte of a set variable, and set element number 7 will be the most significant bit of the same byte. A bit is set to one if the value that it corresponds to is in the set, and zero if it is not.

For example, the set constant [0,3..6,10] would require two bytes of storage. Recalling that two-byte values on the Apple IIGS are stored least significant byte first, the set's binary representation is

```
01111001  00000100
```

Booleans

Boolean variables take on the value of true or false. `True` and `false` are, in fact, predefined boolean constants.

Boolean variables require two bytes of storage each. The ordinal value of a true boolean variable is one, while the ordinal value of a false boolean variable is zero.

Boolean values which are stored in a packed array only require one byte of storage each.

Characters

Characters are members of the ASCII character set. Their ordinal values range from 0 to 127.

Pascal requires that the ordinal values of the digits be sequential. That is, for every character in ['0','1','2','3','4','5','6','7','8','9'], adding one to the ordinal value of the digit must give the ordinal value of the next higher digit, and subtracting one must give the ordinal value of the next lower digit. Pascal does not require the same to be true for the alphabetic characters, although it is in fact true in this implementation. If portability to computers that do not use the ASCII character set is an issue, your program should not depend on the ordinal values of the alphabetic characters being sequential. They are, however, required to be properly ordered, so `ord('z') > ord('a')`.

The first thirty-two characters and the last character in the ASCII character set are not printing characters, so you cannot represent them as character constants.

In strings that do not have a length byte, ORCA/Pascal treats an ASCII zero as an end-of-string character. This allows you to use C-style strings for string-compare and string-copy operations. Note that an ASCII zero embedded in a string will cause a Pascal write operation to stop at the null character.

A character variable requires two bytes of storage, unless it is stored in a packed array, in which case it only requires one byte of storage per array element.

Pointers

Each pointer requires four bytes of storage. It points to a memory location in the heap. The value that the pointer points to is stored in the byte whose address is given by the pointer, and in the bytes that follow if the value requires more that one byte of storage. Pointers are stored least significant byte first.

Pascal has a predefined pointer constant called nil. While the ordinal value of this constant is not defined by Pascal (and, indeed, the concept of an ordinal value is not allowed in Standard

Pascal), it is 0 in this implementation. This fact is especially useful when dealing with the toolbox, which frequently allows a pointer value of zero when passing a pointer value. In all such cases, nil may be used.

Chapter 13 - Derived Data Types

This chapter deals with data types that are derived from those discussed in the last chapter. All of the data types discussed here are represented internally as one or more of the types from Chapter 12.

Enumerations

Pascal allows the declaration of a list of variables that become ordered members of a new data type. For example, you could define a new type

```
color = (red,orange,yellow,green,blue,violet)
```

as the colors of a rainbow. Variables can be defined which have the type `color`. Certain operations can also be performed. The operations are limited to comparisons, assignment, and using the variable (or one of the constants, such as red) as the argument to the `succ`, `pred`, or `ord` functions. Enumerated variables can also be passed as arguments to user-defined procedures and functions, and can be the return type of a user-defined function.

Enumerations are ordered. In the above list, `red < orange`, for example. The ordinal value of the first name in the list is zero, with the ordinal value for each succeeding item increasing by one. Thus, `ord(violet) = 5`. Variables are represented internally as integers. This means that an enumeration can have up to 32768 entries.

Subranges

Subranges specify a limited, sequential range of a scalar data type. The scalar data types include integers, long integers, characters, booleans, enumerations, and other subranges. Whenever a value is assigned to a variable that is declared as a subrange, the value is checked to ensure that it is in the range specified. If it is not, an error will be flagged. The error is occasionally caught during the compile or code generation phase, but in general results in a run-time error. The internal representation and storage requirements for a subrange match those of the base type that the variable is a subrange of.

```
warmcolors = red..yellow;
```

Checking for subrange exceeded errors takes a substantial amount of code and time. For that reason, checking for this type of error is optional at run-time. The {$RangeCheck} directive is used to turn this checking on or off. The {$ISO+} directive will also turn range checking on. Range checking defaults to off.

Unsigned Integers

Subranges of integers and longints get special treatment if all of the allowed values are greater than or equal to zero. In some cases, the compiler can generate more efficient unsigned operations, which can save space and execution time – especially for compares, like the ones typically used in while and repeat loops.

The operations which are faster when unsigned math is used are multiplication, division, modulus, compares other than for equality or inequality (i.e. >, >=, <, <=) and the shift right operation. If both of the operands for these instructions are unsigned, and if peephole optimization is enabled, ORCA/Pascal will use the faster, unsigned version of each operation.

The main issue is detecting when unsigned operations are allowed. The compiler checks each operand to see if it is unsigned, using these rules:

1. For variables, the operand is unsigned if the variable is a subrange with the lower limit greater than or equal to zero. The easiest way to take advantage of this is to declare two types:

   ```
   type
      unsigned = 0..maxint;
      unsigned4 = 0..maxint4;
   ```

   and then to use these types when declaring variables that will never be less than zero. You need to take some care, though, especially with loops. It's easy to make a mistake like

   ```
   var
      count: unsigned;

   begin
   count := 10;
   while count >= 0 do begin
      DoSomething(count);
      count := count-1;
      end; {while}
   ```

   This will cause an infinite loop, since count will drop to -1. As an unsigned value, -1 is interpreted as 65535, and the loop keeps going.

2. For some operations, the result is always treated as an unsigned value. The operations include any value cast to an unsigned integer, as well as the results of the functions abs, sqr, sqrt.

3. For some operations, the result is always treated as a signed value, even if the operands are not signed. This includes subtraction, unary negation, and the result of any function not mentioned in #2.

If you know the result of such an operation is unsigned, you can cast the result to an unsigned subrange to get the benefit of shorter operations for the remainder of the expression. For example, in the expression

```
a := (b-4)*c;
```

if a, b and c are unsigned, and the value of b is known to be greater than or equal to 4, you could use

```
a := unsigned(b-4) * c;
```

to get an unsigned multiply.

4. Constants are treated as unsigned operands if the value is greater than or equal to 0.

Arrays

Arrays are numbered groupings of similar data elements. You can define an array of any data type, including a derived data type. Each array has a subscript type, which is the type of variable used as the index to select from the various array elements. The subscript type can be any scalar type. The array also has a data type, which is the type of each element of the array. This type can be another array, which is how Pascal deals with multiply subscripted arrays.

Arrays are indexed using either integers or long integers, as appropriate. With the small memory model (which is the default), arrays are limited to 64K each, although there is no limit to the total amount of space that can be used by arrays, other than available memory. With the large memory model, the only limitation on the size of an array is available memory.

▲ **Warning** If any array or record exceeds 64K, even if the array or record is allocated dynamically with the new procedure or with NewHandle, you must use the large memory model. ▲

Arrays are stored in memory with the rightmost index incrementing the fastest. For example, for the array

```
matrix: array [1..3,1..3] of real;
```

the elements would appear in memory in this order:

```
matrix[1,1]
matrix[1,2]
matrix[1,3]
matrix[2,1]
matrix[2,2]
matrix[2,3]
```

```
        matrix[3,1]
        matrix[3,2]
        matrix[3,3]
```

The memory requirement for an array is the product of the number of elements in the subscript and the size of an array element. In the above example, `matrix` would require 3*3*4 bytes of memory.

Pascal allows the definition of packed arrays, which tells the compiler that an array should be stored in a way that conserves memory, even at the expense of extra instructions when the array is accessed. In most cases in ORCA/Pascal there is no difference in the internal representation used for packed and unpacked arrays of the same type. Char and boolean arrays are treated specially, however. In unpacked char or boolean arrays, each element occupies two bytes of memory. In packed char or boolean arrays, each element occupies one byte of storage.

Packed arrays of characters can have special meanings and properties. See Strings, later in this chapter, for details.

Strings

In Pascal, strings are a particular kind of array that gets special handling. The general form for a string variable is

```
        packed array[n..m] of char
```

where n is either 0 or 1, and m is greater than 1. In Standard Pascal, n is always 1, and the second is in the range 2..maxint. If you define a string this way, it will be compatible with other implementations of Standard Pascal. ORCA/Pascal extends the ASCII character set to define the character chr(0) to mean the end of the string, allowing strings whose length can change dynamically.

The second form of string uses 0 for the first subscript. When this form is used, m must be limited to the range 2..255. In the toolbox reference manual, strings of this format are described as "Pascal strings." (Curiously, this is despite the fact that Jensen & Wirth, ISO Pascal and ANSI Pascal all define a string the other way. No accepted Pascal standard supports what the toolbox refers to as a Pascal string!)

If you do not care what the internal format is, you can use either form of string. All operations on strings, including the string manipulation procedures and functions, will work on either format. If you do care what format is used, a starting subscript of 0 gives a UCSD string (with a length byte) and a starting subscript of 1 gives a Standard Pascal string (no length byte).

There is a short-hand way to define a string. In the short-hand form, the type is string, optionally followed by a constant in brackets, as in

```
        string[200]
```

The constant specifies the number of characters the string can hold - in effect, this is the maximum subscript range for the packed array of characters. If you use this form, the compiler will generate a string with a length byte for strings whose length is less than 256 characters. For strings with a length of 256 or more characters, a length byte is no longer possible, and the compiler automatically switches to Standard Pascal strings.

There is one place where the specific format of the string may be important to you. When you extract the address of a string, as in the QuickDraw call

```
DrawString(@'Hello, world.');
```

the compiler returns a pointer to a length byte, followed by the ASCII characters, and then a null terminator. For most toolbox calls, the address operator returns what you want: a pointer to a string with a length byte. If you need a C string (null terminated), you can get it by adding 1 to the address, as in

```
DrawCString(pointer(ord4(@'Hello, world.')+1));
```

When you pass a string as a parameter to a function or procedure, you must insure that both the parameter and the string are in the same format. String constants are automatically coerced to the appropriate format.

Records

Records are collections of unlike data elements. Like arrays, records appear in memory as a series of primitive data types.

Pascal allows the definition of a variant record. A variant record is a record that can contain different kinds of data at different points during the program's execution. Pascal also allows the `new` procedure, which sets aside space for a dynamic variable from a heap, to be called with or without a list that specifies the structure for a given variant record. ORCA/Pascal takes advantage of the knowledge available when `new` is called, and sets aside only enough space to satisfy the request. For example, for

```
v: record
   case realvar: boolean of
      true: (r: real);
      false: (i: integer);
      end;
```

the call

```
new(v,true);
```

would allocate six bytes of memory - two for the variant selector and four for the real variable. However, the call

```
new(v,false);
```

allocates only four bytes of memory. This can be very useful, but leaves the possibility of an assignment like

```
        new(v,false);
        v.r := 1.0;
```

which can lead to disastrous results. If the variable will hold more than one kind of data, don't specify the variant part in the call to new. That way, the largest amount of memory that can ever be used will be allocated. For example,

```
        new(v);
```

allocated six bytes of storage, so that either an integer or real can be stored.

When a variant record is defined, it is legal to omit the tag variable. For example, if you will not be setting or reading the variable realvar from the record shown earlier, the variable can be left out. This changes the internal format of the record: since the variable is not set or used, no space is reserved for it. Without a tag variable, the variant record looks like this:

```
        v: record
           case boolean of
              true: (r: real);
              false: (i: integer);
              end;
```

Instead of four or six bytes of storage, this record will require two or four bytes.

Under ORCA/Pascal, if you are using the small memory model, each record must use less than 65536 bytes of memory. The only limit on the total amount of memory used by all records is the available memory in your computer. If you are using the large memory model, the only limit on the size of a record is available memory.

▲ **Warning** If any array or record exceeds 64K, even if the array or record is allocated dynamically with the new procedure or with NewHandle, you must use the large memory model. ▲

Files

File variables represent sequentially accessed lists of variables. Each of the variables in a file must be of the same type, but unlike an array, the values cannot be accessed in a random order. A file is always available for reading or writing, but in Standard Pascal, never both. When reading from a file, you start by reading the first element and continue until the last element is read. When writing to a file, you always start with an empty file and add values to the list.

In ORCA/Pascal, files are always stored on disk. If no file name has been given when the file is reset or rewritten, a name is assigned by forming a file name with the characters "SYSPAS" and a four-digit number, starting with "SYSPAS0001". Each time a new file variable is reset or rewritten, the number is incremented. These files will end up on the work prefix. They are not deleted when program execution ends, so you can access them later. Files declared with the predefined file type `text` will show up as ProDOS TXT files, while any other file type is a binary (BIN) file.

ISO ORCA/Pascal supports some extensions that allow more flexible use of files. See `seek` for a way to access file elements randomly, and `open` for a way to open a file for both input and output. These are discussed in Chapter 22. You also have complete access to the primitives available from ProDOS. Δ

Chapter 14 – Object Pascal

ISO All of the features discussed in this chapter are extensions to Standard Pascal. They are almost identical to the object extensions found on Macintosh Pascal compilers, but if you will be porting your program to any other platform, you should avoid these features. Δ

Object Oriented Programming

ORCA/Pascal supports object oriented programming, implementing the object extensions used by Apple Computer on the Macintosh. There are three advantages to using this proven object technology:

1. Since the object extensions and underlying Pascal language are almost identical on the Macintosh and Apple IIGS, you can learn object oriented programming from Macintosh Pascal books.
2. The Macintosh toolbox and Apple IIGS toolbox are very similar in overall design and implementation, even though there are many differences in individual calls. Using the same object extensions makes it easier to port programs between the Macintosh and Apple IIGS. In fact, with some carefully designed objects that hide the differences between the tools on the two computers, you could write the same program and simply compile it on the different computers.
3. Because we're using a proven design that has worked well for years, you can feel confident that the object oriented extensions you see in ORCA/Pascal are robust and complete.

The other chapters in this portion of the ORCA/Pascal reference manual assume you know Pascal well, and are looking up specific information. This chapter does not assume you already know Object Pascal. If you refer to an individual section in the chapter, you will find all of the information you need about a particular feature of the compiler, but it will generally be a little wordier than other parts of the language reference. If you read this chapter front to back, you'll get a quick but complete overview of Object Pascal and how it is used.

Object Pascal is really two things, though. This chapter describes the mechanics of the Object Pascal extensions, but to use them effectively, you need to learn some of the ideas behind object oriented programming. There are many books available that teach the design principles of object oriented programming. The Macintosh Pascal books are one good source of information. There are also a number of books that teach the ideas behind object oriented programming without dealing with a specific language. If you intend to use these extensions, it would be a good idea to get one or more of these books to learn some of the ideas, and get tips on good object oriented program design.

Objects

This section describes the mechanics of declaring objects. An object is a data type that, superficially at least, looks and works like a record. This section starts with the definition of a record, and gradually introduces objects by discussing the various differences between objects and records.

Objects

Like a record, and object has zero or more fields. A simple object type declaration looks exactly like a record with the reserved word `object` used instead of the reserved word `record`.

```
type
   box = object
      top, bottom, left, right: integer;
      end;
```

Unlike a record, an object type can only be declared as a type; you cannot create an object variable and place the type right after it. For example, this definition is legal for a record, but the same form cannot be used for an object:

```
var
   box: record  {objects cannot be defined like this}
      top, bottom, left, right: integer;
      end;
```

Object types also have to be declared at the program level of a program, or in the implementation or interface part of a unit. Object types cannot be defined in the type part of a procedure or function.

There's a certain amount of terminology that goes along with object oriented programming, and it helps to match those terms up with what you are doing in object Pascal so you can understand what the object oriented programming books, magazines, and so forth are talking about. The object type is actually called a **class**, not an object. The **object** is more closely associated with the variable, as opposed to the type. As you'll see in a moment, though, more than one variable can refer to the same object. Fields in an object are called **instance variables**.

Object Variables

An object is defined just like any other variable. Objects are not restricted to the program level, like the class is. Unlike records, though, objects must be initialized before they can be used. We'll look at how objects are initialized in the section "Allocating Objects with New," later in this chapter.

```
var
   abox: box;
```

The object abox is called an **instance** of the class box. It's also proper to call the object a **member** of the class, although this term can apply to some other situations, too.

Accessing Instance Variables in Objects

You use and assign values to the instance variables in an object as if it were a field in a record. All of the same rules apply, and the dot operator is still used to dereference the instance variables.

Methods

A **method** is a procedure or function which is declared in the class and defined later in the program. The declaration of the methods must follow the declaration of all of the instance variables in the class. With the exception of where it is located, a method declaration looks and works just like a procedure or function declaration in the interface part of a unit.

Here's the box class, with three methods added. We'll see how to define the method in "Defining Methods," later in this chapter.

```
type
   box = object
      top, bottom, left, right: integer;
      function Area: integer;
      procedure Fill (ptop, pleft, pbottom, pright: integer);
      procedure Grow (size: integer);
      end;
```

Adding methods to an object gives us a neat, tidy, portable way to collect all of the information about a class into one place. While it's possible to access the instance variables of an object from anywhere in a program where the object itself can be accessed, in general, only the methods in a class use the variables. In this case, we've defined three methods. Fill is used to set up the top, bottom, left and right sides of the box. Area returns the area of the box. If we're careful to follow the rule that only the methods use the instance variables, we can set up Fill so it always makes sure left is less than right and top is greater than bottom, then use those assumptions in Area to make that method more efficient. Grow is a method that expands the size of the box by multiplying each instance variable by `size`.

Inheritance

You can create a class that is based on an existing class. The process of creating a class based on one that already exists is called **inheritance**. The original class is placed right after `object`, and is enclosed in parenthesis. All of the instance variables and methods from the original class become a part of the new class. You can add new instance variables and new methods; it's also possible to replace a method with a newer one.

Here's the framework for a new class, based on the class box that we've been using throughout this chapter.

```
cube = object (box)
   front, back: integer;
   function Volume: integer;
   end;
```

Cube is called a **descendent** of box; it's also a **subclass**. Box is the **ancestor** of cube; it's also called the **superclass**. Since box doesn't have any ancestors, it's also called a **root class**.

This class still has all of the original instance variables and methods from box, so you can set and read a total of six instance variables (top, bottom, left, right, front and back). All of the original methods are still there, too, along with the new one, Volume. This is a very powerful concept, since it lets us create something called a box, and define and work on it in a sensible way, then go back and define a new class that's sort of like a box, but with some new features. That's really what a cube is, too: a box with a new dimension. We did this by adding two new variables and a new method, none of which were needed, or made any sense, for a box.

Overriding Methods

In the last section, we looked at cube, a subclass of box. Cube was created by adding two variables and a method to an existing class. The problem is that two of the old methods don't quite work the same way for a cube and a box. Area still works, and makes sense, but Fill needs to handle six variables instead of four, and Grow will need to deal with the new instance variables, too.

Replacing existing methods with a different version of the method is called **overriding** the method. To override the method, declare it just like it was new, but follow it with the directive override, like this:

```
cube = object (box)
   front, back: integer;
   function Volume: integer;
   procedure Fill (ptop, pleft, pbottom, pright,
                   pfront, pback: integer); override;
   procedure Grow (size: integer); override;
   end;
```

Methods

Defining Methods

For the most part, defining a method works just like defining a procedure in the implementation part of a unit when it has already been declared in the interface part. In both cases, the definition doesn't list the parameters or function return results a second time. In fact, the only difference is the name. The method name is a combination of the class name, a period, and the name of the method itself. It looks sort of like a record with a field access. This naming convention is needed so you can distinguish between methods that have the same name, but are declared in two different objects. It's also perfectly legal to have a normal procedure or function with the same name as a method.

Here are some of the methods for the box and cube classes. If you look closely, you'll see that these methods are using the instance variables from the class without using the class or object name; we'll discuss why that works in "The Self Object," later in this chapter.

```
function box.Area;

begin {Area}
Area := (top-bottom)*(right-left);
end; {Area}

procedure box.Fill;

begin {Fill}
if ptop < pbottom then begin
   top := pbottom;
   bottom := ptop;
   end {if}
else begin
   top := ptop;
   bottom := pbottom;
   end; {else}
if pright < pleft then begin
   right := pleft;
   left := pright;
   end {if}
else begin
   right := pright;
   left := pleft;
   end; {else}
end; {Fill}
```

```
procedure box.Grow;

begin {Grow}
top := top*size;
bottom := bottom*size;
left := left*size;
right := right*size;
end; {Grow}

function cube.Volume;

begin {Volume}
Volume := (top-bottom)*(right-left)*(back-front);
end; {Volume}
```

All of the normal rules for defining procedures and functions apply to methods, too. You can, for example, define new procedures and functions that are nested within the method.

Note Macintosh compilers allow the parameter list to be listed a second time, so long as the parameter list exactly matches the one in the class. They also allow the parameter list to be omitted. ORCA/Pascal follows the ideas for forward declared procedures in Standard Pascal, which does not allow the parameter list to appear a second time.

Inheriting Previous Methods

In our example, cube contains two methods that override methods in its ancestor, box. When we define the methods for cube, we have a choice. We can create the method from scratch, just like we did for `cube.Volume` in the last section, or we can inherit the previous method, then add any new statements, like this:

```
procedure cube.Fill;

begin {Fill}
inherited Fill(ptop, pleft, pbottom, pfront);
if pback < pfront then begin
   back := pfront;
   front := pback;
   end {if}
else begin
   back := pback;
   front := pfront;
   end; {else}
end; {Fill}
```

```
procedure cube.Grow;

begin {Grow}
inherited Grow(size);
front := front*size;
back := back*size;
end; {Grow}
```

Using the inherited method lets us reuse the older method instead of duplicating all of the effort. There's another even more important advantage to inheriting the older method that really doesn't become obvious until you're using objects that were written a long time ago, or perhaps by someone else: you don't have to know anything at all about the old method, other than what it does. *How* it does what it does is not your problem, and in fact, the original author can make changes and improvements to the original method, and you get the advantages of those improvements by just relinking your program. If that sounds sort of like a desktop program getting the advantages of a new version of the toolbox, it is – and that's one of the reasons object oriented programming and toolbox programming fit together so well.

Using Objects

Allocating Objects with New

Before looking at how objects are actually used, it might help to understand a little about how they are implemented internally. When you define an object variable, as in

```
var
   abox: box;
```

the compiler sets aside four bytes of storage. It's quite literally a pointer to the object, not the object itself. Just as with a file, which is also a pointer to a value, you have to take steps to initialize the object – in this case, you have to allocate the memory with the new procedure:

```
new(abox);
```

New sets aside space for the object, and does some internal initialization so you can call methods. It does not initialize the instance variables in any way.

Note Macintosh compilers implement objects as handles, not pointers.

Disposing of Objects with Dispose

Just as with any other variable allocated with new, it's up to the program to dispose of the object when it is no longer needed, like this:

```
dispose(abox);
```

Passing Messages to Objects (Calling Methods)

Once an object has been allocated with new, you can call the various methods. The correct object oriented term for calling a method is passing a **message** to the object. Here's the body of a simple program to set up a box object and calculate the area, before and after growing the object:

```
begin
new(abox);
abox.Fill(10, 10, 0, 0);
writeln(abox.Area);
abox.Grow(2);
writeln(abox.Area);
dispose(abox);
end.
```

The Self Object

Methods are designed to work on the instance variables in an object, but there is no obvious way for the method to know *which* object it is supposed to work on. This is actually handled with the self variable, which is predefined in every method. The self variable is the object that the message is being sent to. In the case of

```
abox.Grow(2);
```

the message is being sent to the object abox, so self refers to abox. It's as if there is an implied parameter for every method, and that parameter is the object. The method itself has an implied with statement, so you don't have to type `self.top` to get at the instance variable top. In effect, a method acts like it is defined this way:

```
procedure cube.Fill (self: cube);

begin {Fill}
with cube do begin
   ...
   end; {with}
end; {Fill}
```

All of the implications of defining this with statement apply, too. For example, the cube class defined an instance variable called `front`. If the method defined a variable called `front`, it will be impossible to access that variable, since the compiler would always treat `front` as `self.front`.

You can use the self parameter explicitly, too. It's legal, for example, to write Grow this way:

```
procedure cube.Grow;

begin {Grow}
inherited Grow(size);
self.front := self.front*size;
self.back := self.back*size;
end; {Grow}
```

Assigning Objects

When you assign one object to another, internally, the compiler actually assigns the pointer to the object, not the object itself. Thinking of this in terms of records, it's like assigning a pointer to a record to another pointer to a record, not like assigning one record to another. What you end up with, in object oriented programming terms, is two instances of the same object.

For example, let's define a program that uses two instances of the class box.

```
var
   abox, bbox: box;

begin
new(abox);
abox.Fill (10, 10, 0, 0);
writeln(abox.Area);
bbox := abox;
```

One thing that should jump out is that we never used new to initialize bbox. Thinking about how objects are handled internally, this makes sense: we've assigned a pointer to an object that has already been initialized, so there is no need to call new a second time. Continuing with the program:

```
abox.Grow(2);
writeln(abox.Area);
writeln(bbox.Area);
```

Both of these writeln statements will print the same value, since abox and bbox are instances of the same object.

```
dispose(abox);
end.
```

There is no need to dispose of both abox and bbox, since they refer to the same object. In fact, disposing of the object twice isn't even legal, and would generally lead to a crash or corrupted memory.

You can make a copy of an object; see "Copying Objects with Clone," later in this chapter, for details.

It's also possible to assign objects that aren't from the same class, as long as the object being assigned is a descendent of the object being assigned to. Assuming cube is a descendent of box, all of these assignments are legal:

```
var
   abox, bbox: box;
   acube, bcube: cube;

begin
new(abox);
bbox := abox;
new(acube);
bcube := acube;
abox := acube;
```

Finally, you can assign nil to an object:

```
bcube := nil;
```

Files of Objects

It is legal to create a file of objects, but keep in mind that the objects themselves – in other words, the pointers – are written, not the instances of the objects. If you want to write the instance variables, you need to write them individually.

tObject

Assigning objects creates two instances of the same object. There are many cases where you really need to create a copy of the original object, instead. The library object tObject can do this; it also has facilities for disposing of objects.

tObject is defined in the unit ObjIntf, and you do need to put a uses statement in your program to get access to this object. The uses statement is

```
uses ObjIntf;
```

The object itself is defined like this:

```
type
   tObject = object
      function ShallowClone: tObject;
      function Clone: tObject;
      procedure ShallowFree;
      procedure Free;
      end;
```

Copying Objects with Clone

When you send a Clone message to an object, it makes a copy of the object and returns the copy. To use Clone, you have to declare your own classes as subclasses of tObject. Here's a previous example, rewritten to use tObject to make a copy of the object. Compare how this works with the original example, which used an assignment statement instead of sending a Clone message.

```
uses
   ObjIntf;

type
   box = object (tObject)
      top, bottom, left, right: integer;
      function Area: integer;
      procedure Fill (ptop, pleft, pbottom, pright: integer);
      procedure Grow (size: integer);
      end;

   {define the methods here}

var
   abox, bbox: box;

begin
new(abox);
abox.Fill (10, 10, 0, 0);
writeln(abox.Area);
bbox := box(abox.Clone);
abox.Grow(2);
writeln(abox.Area);
writeln(bbox.Area);
dispose(abox);
end.
```

In this example, the area printed for bbox will be different from the area printed for abox, since there are two different object involved, and only the object abox has grown.

Disposing of Objects with Free

The Free method gives you another way to dispose of an object. Free is generally used when you need to do more than just dispose of the object. For example, if one of the instance variables in an object is a window pointer, you might want to close the window as a part of disposing of the object. By overriding Free with your own version, you can easily add the code to dispose of the window.

That may seem like a lot of trouble, but it helps stick with the object oriented programming paradigm. If you use Free to dispose of all of your objects, your methods don't have to deal with any special cases, or even know if any special cases exist. Whether there is any extra work to do when disposing of an object or not, Free will always do the job completely and accurately.

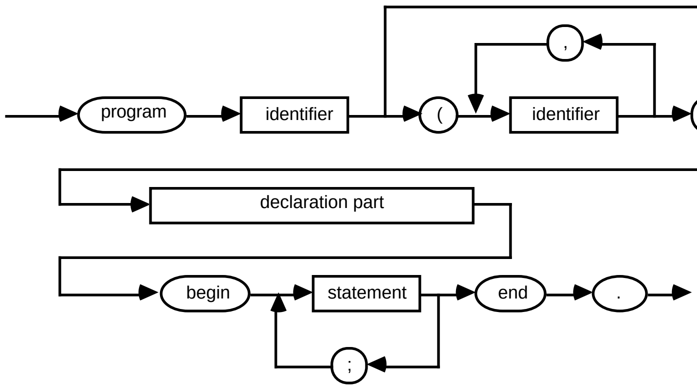
ShallowClone and ShallowFree

Clone and Free are there for your program to override. You can override Clone to do additional initialization on objects, like opening new windows. You can override Free to do additional work before disposing of an object, like closing windows.

ShallowClone and ShallowFree do the same thing as Clone and Free, but you should not override these methods. These methods are the primitives. You should leave them available and unmodified, so you have a safe way to create a new object or dispose of an old one when you need to avoid any additional initialization.

Chapter 15 - The Program

program



A Pascal program consists of a program header, a declarations section, a block, and a period. The program header consists of the reserved word `program`, followed by an identifier that is the program name, a possibly empty list of external variables separated by commas and enclosed in parentheses, and a semicolon. The format for the declarations section and block are covered in future chapters.

The name of the program is defined at the program level. It has no further significance, and can be reused later in the program. External variables are variables whose value is assigned at program execution time. The mechanism is left unspecified by Pascal, and in fact no mechanism is required to exist. In ORCA/Pascal, with the exception of the predeclared file variables `input`, `output` and `errorOutput`, there is no difference between a program that declares variables in the program header and one that does not.

Two predeclared file variables are required in Standard Pascal, `input` and `output`. In addition, ORCA/Pascal supports `errorOutput`, which functions exactly like output, except that the characters are sent to the Apple IIGS error output device, rather than the standard output device. If these files are used in a program, they must appear in the program header. When `input` is included in the program header, it becomes defined as a text variable. It is implicitly initialized with a `reset(input)`. Further applications of `reset` to the input file have no effect. In most cases, characters read from `input` will come from the keyboard.

The second predeclared file variable is `output`. If output appears in the program header, a `rewrite(output)` is generated by the compiler. Further applications of `rewrite` to the output file have no effect. Output generally goes to the text screen, but like input, output can be redirected.

ISO The file `erroroutput` is an extension to Standard Pascal. Δ

The last predeclared file variable is `erroroutput`. It functions exactly like `output`, except that characters sent to `erroroutput` are passed to the Apple IIGS error output device, rather than the standard output device.

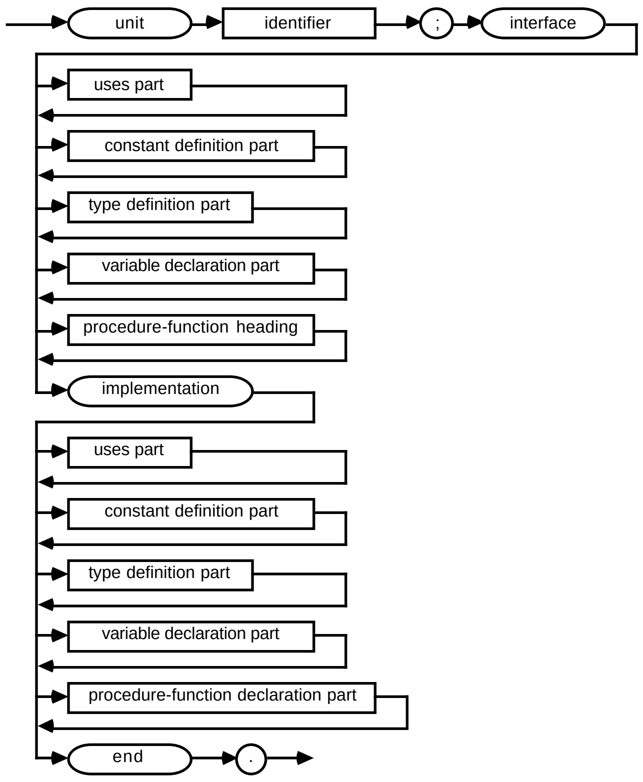
With the exception of `input`, `output` and `erroroutput`, all variables listed in the program header must be declared in the declaration part of the program block.

Example programs:

```
{the smallest legal Pascal program}
program s;begin end.


{the classic first program}
program greetings(output);

begin
writeln('Hello, world...');
end.
```

Chapter 16 - Units

unit



ISO Units are an extension to Standard Pascal. They are not available on many mainframe implementations of the language. Δ

UCSD Pascal made several additions to Standard Pascal. One of the most universally adopted is the unit. Like programs, units consist of declarations of constants, types, variables, procedures and functions, but unlike programs, a unit cannot be executed. Instead, units are used to create libraries, or to break large programs up into smaller, modular collections of similar procedures and functions. While a unit cannot be executed, it is certainly possible to create a program which executes procedures and functions from the unit.

A unit consists of three parts. The first part is the unit header. The header starts with the reserved word `unit`, and is followed by the name of the unit and a semicolon. Like the name of a program, the name of the unit has no significance in the rest of the source file.

The next part of the unit is the interface part. The interface part starts with the reserved word `interface`. It contains a constant, type and variable section that is coded exactly like the corresponding sections in a program. All of these may be accessed from the remainder of the unit, or from any unit that uses the unit where they are defined. The remainder of the unit consists of procedure and function headers. These headers are coded exactly like forward declared procedures in a program, with the exception that the directive forward is omitted. Externally declared procedures, tool interfaces and ProDOS interfaces may also be declared in this section.

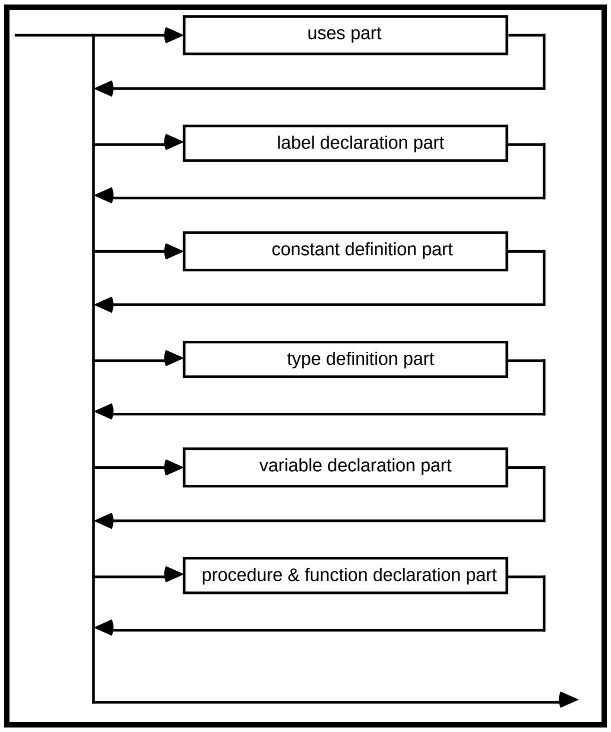
Immediately after the interface section is the implementation section. It starts with the reserved word `implementation`. The implementation section consists of a constant section, a type section, a variable section, and procedures and functions. These sections follow the same rules as the corresponding sections in a program. Any procedure or function defined in the interface part that was not defined as external, a tool, or a ProDOS interface must appear in the implementation part. As with the definition of a forward declared procedure, the parameter list and function return type are not repeated in the declaration. Any constants, types or variables that appear in the implementation section, and any procedures and functions whose header did not appear in the interface part, can only be used from within the unit. Their names are hidden from other units and from the program that uses the unit. It is even possible, for example, to create two units, each of which has a procedure or function with the same name, so long as all but one of the procedures or functions appears only in the implementation part.

The structure of units is such that a program using units resembles a tree. While a unit can be, and often is, used by more that one other unit in a complex program, there are cases when a procedure or function is needed in a unit that is used by the unit where it is defined. For example, unit A may declare a procedure called GotoXY, but GotoXY may need some output routine defined in unit B. If unit B has a procedure that calls GotoXY, the structure of the unit starts to break down. There are several programming strategies that can be used to handle this situation. First, the two units can be combined. From the standpoint of creating a well-structured set of units, this is usually the best solution. The second possibility is to create a third unit which has GotoXY and the procedure or function from unit B that GotoXY calls. This solution usually works best with constants, types and variables. In a program that consists of multiple units, there is generally one unit (we call it Common) that contains constants, types, variables and a few low-level procedures and functions that are used throughout the program. As a last resort, the procedure used by GotoXY can be declared as an external procedure in the implementation part of unit A. So long as a procedure or function appears in the interface part of some unit, or in the main program, it can be defined as external from any other part of the program, and used successfully. The only caveat is to insure that the external definition is in the implementation part of the unit. If the external definition appears in the interface part, the definition may conflict with the definition of the actual procedure.

For samples of units that are used as toolbox interface files, see the TOOL.INTERFACE folder of the extras disk. See the section on the uses statement in the next chapter for details on how to use a unit from another unit or from a program.

Chapter 17 - The Definition Section

definition section



The first part of a block is the definition section. It is here that the variables, constants, types, labels, procedures and functions that are used by the block are defined. This chapter discusses all of these except for the declaration of procedures and functions - those are discussed in the next chapter.

One point worth noting is that the order of the parts in the header of the block must not be changed. While procedures and functions can be mixed, labels, if used, must come before any other part of the header; constants, if used, must come next; and so on.

Uses

ISO The uses statement is an extension to Standard Pascal. It is used to access interfaces to units which have already been compiled. Units are covered in Chapter 16. ∆

The uses statement must appear before any other declarations in the block, including declarations of labels. After the uses keyword, a list of the units to use appears, with multiple units separated by commas. The statement ends with a semicolon. Each of the units listed is opened, in turn, and the declarations from that unit are processed.

It is possible for a unit to include a uses for another unit. For example, a unit can be defined which contains all of the types and constants that are used in the other portions of a separately compiled program. A simple example is a unit that declares several string sizes:

```
unit Common;

interface

type
   string20  = string[20];
   string80  = string[80];
   string255 = string[255];
   stringMax = string[maxint];

   string255Ptr = ^string255;
   stringMaxPtr = ^stringMax;

implementation

end.
```

Another unit in the program might declare a string search function which accepts two parameters, a pointer to a target string of type string255, and a pointer to a string to search of type stringMax. This unit might look like this:

```
unit Strings;

interface

uses Common;

function Find (target: string255Ptr; buffer: stringMaxPtr): integer;
```

```
implementation

function Find;

begin
{code for Find goes here}
end;

end.
```

If another part of the program will be using the strings unit, the common unit must also be listed in the uses statement, and must appear before the strings unit. This is because the interface part needs the type definitions from the common unit, which are not available unit after the header for the common unit has been processed. If the strings unit is used before the common unit, an error will result. The correct statement, then, for a unit or program that needs access to the strings unit, is

```
uses Common, Strings;
```

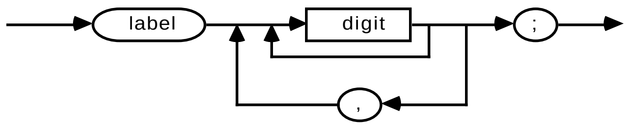
More than one unit may be used in the same block by coding one uses statement immediately after another. The previous example can also be coded as

```
uses Common;
uses Strings;
```

When you use a unit, the compiler actually looks for an interface file created when the unit was compiled. The compiler begins its search by looking in the library prefix, using the pathname 13:ORCAPascalDefs. If the interface file is not found there, the compiler continues the search with the default prefix, 8:. You can use the LibPrefix directive to tell the compiler to look in some other folder.

Labels

label declaration part



Labels are used as the destination for the goto statement. They have no other use in Pascal. Any label used in a block must be declared in this part of the header. Each label is an integer ranging from 1 to 9999. A label cannot be declared twice in the same header.

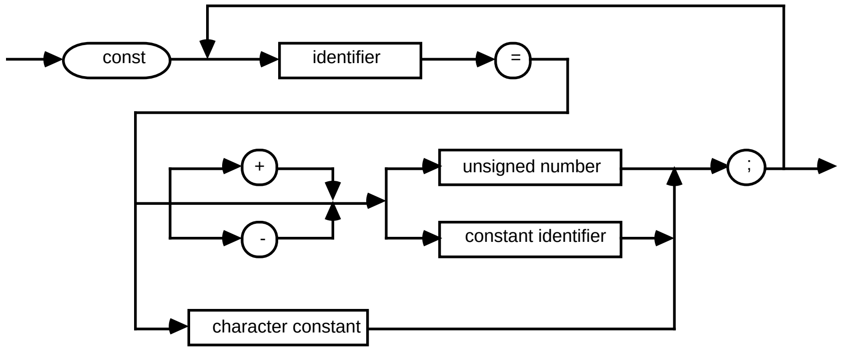
If a label is listed in the header, it must occur in the block.

Example:

```
lab
   1,03,999;
```

Constants

constant definition



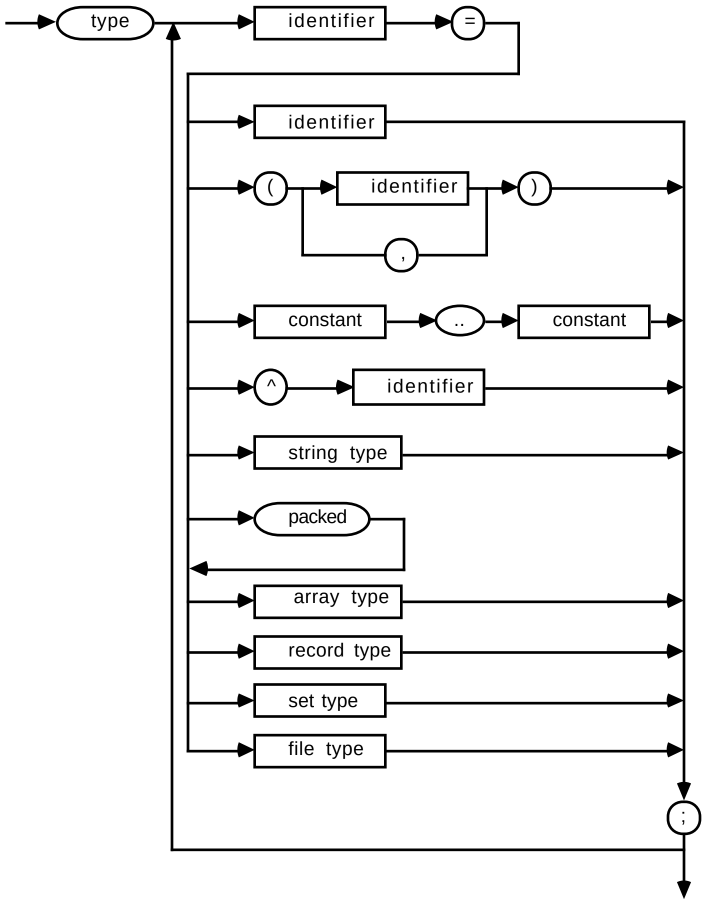
The constant part of the header is used to assign names to values used in the statement part of the block. Constants can be defined for any scalar type, including reals, integers, characters, booleans, subranges and enumerations. Constant strings are also valid. A leading + or - sign is valid on real and integer constants, but no other operations are allowed. You cannot, for example, define a constant as another constant plus one. Internally, integer constants are stored in two bytes if the specified value is within the range -maxint to +maxint. An integer constant which is less than -maxint or greater than +maxint will be stored in four bytes. An integer constant which is not within the range -maxint4 to +maxint4 will be flagged as an error by the compiler. All real constants are represented internally in the SANE 10-byte extended format.

Example:

```
const
   one            = 1;
   minusone       = -one;
   pi             = +3.141593;
   filename       = 'myfile';
   failing        = 'F';
   sky_is_falling = false;
   best_color     = red;
```

Types

type declaration part



The type section allows you to define a type that can be used later to declare variables or other types. An identifier used as a type can be any of the types provided by ORCA/Pascal, or it can be a user-defined type. The predefined basic types in ORCA/Pascal include integer, longint, byte, char, real, double, extended, comp and set. The predefined derived types in ORCA/Pascal are enumerations, subranges, arrays, records, objects and files.

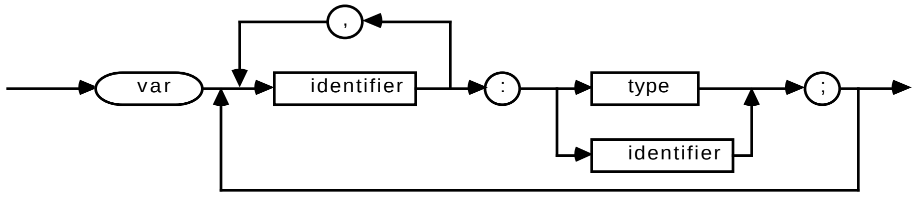
The type string is not really a new type; rather, it is mnemonic for `packed array [0..n] of char`, where `n` is greater than one and less than 256, or `array [0..n] of char` when `n` is greater than 255. The syntax for specifying variables of type string is `string[size]`. If size is omitted, then the string is defined to be of length 80 characters.

Example:

```
type
   name = string[20];
   House = record
      bedrooms: integer;
      baths: integer;
      den,kitchen,dining_room: boolean;
      price: real;
      end;
   matrix = packed array[1..10,1..10] of real;
   address = integer;
   phone_number = record
      area_code, prefix: 0..999;
      number: 0..9999;
      end;
   car = (Ford,GM,Iococa);
```

Variables

variable declaration



The variable part is used to declare the variables which are used in a block. The variables are not initialized in any way - to be sure of the value they contain, you must assign them a value. Variables exist for the length of the activation of the block. This means that if you call a procedure once and assign values to variables declared within that procedure, the variables may not have the same values originally assigned when the procedure is called a second time.

Example:

```
var
   i,j,k: integer;
   my_car: car;
   my_residence: record
      myhouse: house;
      myphone: phone;
      end;
```

External Variables

ISO External variables are an extension to Standard Pascal. Δ

ORCA/Pascal supports external variables. To create an external variable, place the word `extern` or `external` right before the type of the variable, like this:

```
i,j,k: extern integer;
```

This tells the compiler that the variables (in this case, i, j and k) are available, but the compiler will not reserve space for the variables. Instead, the compiler expects the linker to find the proper variables, just as it would for an external procedure or function.

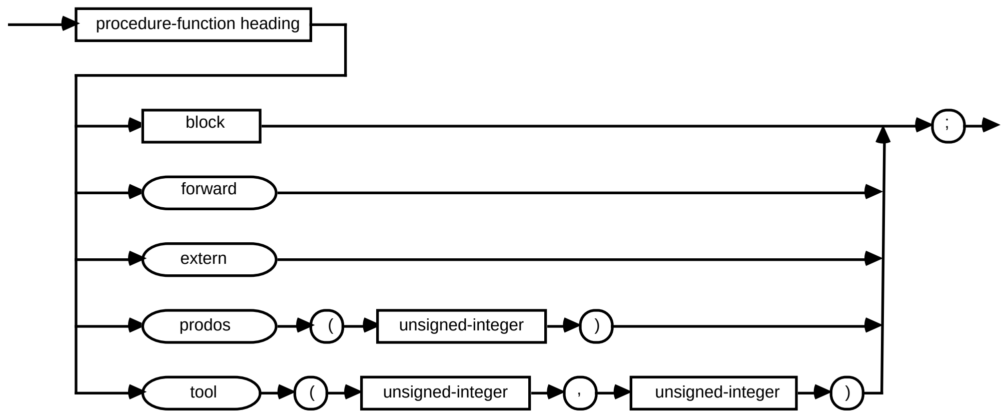
The variables themselves can be defined several ways. From Pascal, the variables could be defined in a unit or in the main program. If the variables are defined in a unit, they must be defined in the interface part of the unit, not the implementation part, since the linker will only use variables from another unit that appear in the public part of that unit. From C, the variables should be declared as normal, external variables, but the names of the variables must not contain any lowercase letters. From assembly language, the variables must be global, so the names must be declared with a START or ENTRY directive.

There are two common uses for external variables. The first is to gain access to a variable that was declared in a unit that you either can't access with a uses statement, or don't find it convenient to access that way. For example, if unit A uses unit B, and unit B needs access to a variable in unit A, you could declare the variable as external in unit B. (Of course, an even better solution is to move the variable to unit B.)

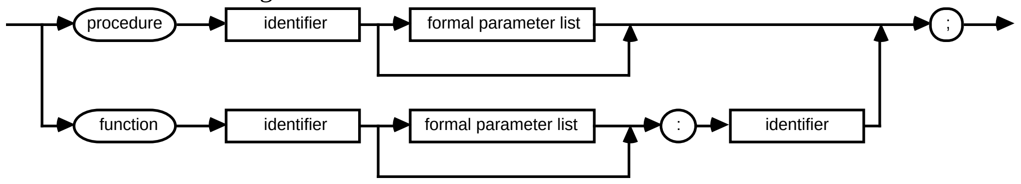
The other common use for external variables is for initialized tables, which are sometimes easier to declare in assembly language or C. In that case, declaring the variable as external gives the program direct access to the variables from the other language.

Chapter 18 - Procedures and Functions

procedure/function declaration



procedure/function heading



Procedures and functions are defined in the header part of a block, right after variables. Procedure and function declarations can be mixed in any order you choose. As seen from the syntax diagrams above, each procedure or function has a name, an optional parameter list, and its own declaration and statement part. The declaration part can contain more variables, constants, types, labels, procedures and functions. Any identifiers declared here are available only within the local block. Identifiers declared before the procedure or function are available within the procedure or function, so long as the identifier is not redeclared. In that case, the local definition has precedence.

For example, consider this procedure:

```
procedure nest;

var
   a,b: integer;
```

```
      procedure inside;

      var
         b,c: integer;

      begin
      {code}
      end;

   begin
   {code}
   end;
```

The variable a can be accessed from the statement part of both procedures. Since c is declared in inside, it can only be accessed from there - it is not available from the procedure nest. Finally, b is declared in both procedures, so both procedures can use a variable called b, but it will not be the same variable - the variable defined in nest is different from the one defined in inside.

Value Parameters

Pascal is capable of passing variables by value or by reference. When a variable is passed by value, the parameter is declared as if it were in a var declaration part of a header. The variable is then available within the procedure as if it were declared locally to that procedure. Any changes made to the variable in the procedure have no effect on the value passed when the procedure was called.

For example, the following function can be used to make sure a character is uppercase only. Note that this function assumes that we are using the ASCII character set. This is true for ORCA/Pascal, but may not be true for Pascal compilers on other computers.

```
function upper(ch: char): char;

begin
if (ch >= 'a') and (ch <= 'z') then
   ch := chr(ord(ch)-ord('a')+ord('A'));
upper := ch;
end;
```

Despite the fact that the variable ch is changed within the function, it does not change the value of the variable which corresponds to ch in the calling procedure. If the above function is called like this:

```
for i := 1 to linelen do
   write (upper(line[i]));
```

then the values of the characters in line remain unchanged.

Variable Parameters

The only difference between the definition of a value parameter and a var parameter is that the var parameter is prefixed by the reserved word `var`. Rewriting our function from above by using a var parameter:

```
function upper(var ch: char): char;

begin
if (ch >= 'a') and (ch <= 'z') then
   ch := chr(ord(ch)-ord('a')+ord('A'));
upper := ch;
end;
```

If we called upper the same way we did before, the characters in `line` would be shifted to uppercase. Since var parameters must be objects whose values can be changed, you must pass a variable, not the result of an expression, when using a var parameter.

Passing Procedures and Functions as Parameters

Pascal allows procedures and functions to be passed as parameters to other procedures and functions. There are some restrictions that must be observed when doing this:

1. The procedure or function being passed must be declared at the program level. That is, it must be declared in the header section of the program block, not in the header of another procedure or function.
2. Predefined procedures and functions cannot be passed as parameters. These include Standard Pascal procedures and functions, as well as those provided with ORCA/Pascal.
3. Tool calls and ProDOS calls cannot be passed as parameters.
4. The parameter list for the procedure or function must match the declared parameter list exactly.

Unlike the older Jensen & Wirth Pascal, ISO Pascal requires that the parameter list be specified for a function or procedure passed as a parameter.

The last of the above requirements is very strict. It means that the parameter lists must be identical in form, and that the parameters must be type compatible as well. For example, if a procedure is declared as

```
procedure proc(a: integer; b: integer; c: integer);
```

then it cannot be passed as a parameter to this procedure:

```
procedure nope(procedure pass (i,j,k: integer));
```

As an example of a practical application of this capability, here is a program that uses a Pascal function to integrate a mathematical function. Rather than writing the integration procedure twice, we write it once and pass the function to integrate as a parameter.

```
{$keep 'stuff'}
program demo(output);

  function f1(x: real): real;
  begin
  f1 := sqrt(abs(x));
  end;

  function f2(x: real): real;
  begin
  f2 := exp(x/2.0);
  end;

  function integrate(a,b: real; steps: integer;
                     function f(x: real): real): real;

  {Trapezoidal integration}
  var
    i: integer;                         {loop variable}
    sum,                                {area under curve so far}
    x,                                  {center of current trapezoid}
    dx: real;                           {width of a trapezoid}

  begin
  sum := 0.0;
  dx := (b-a)/steps;
  x := a + dx/2.0;
  for i := 1 to steps do begin
    sum := sum + f(x)*dx;
    x := x + dx;
    end;
  integrate := sum;
  end; {integrate}

begin {demo}
writeln (integrate(0.0, 1.0, 50, f1));
writeln (integrate(0.0, 10.0, 100, f2));
end.
```

Univ Parameters

ISO Univ parameters are an extensions to Standard Pascal. ∆

ORCA/Pascal supports a method of bypassing type checking called a universal parameter. When you define a procedure or function, simply place the key word `univ` before the type of the parameter. The compiler will then accept anything as a parameter that is the same size as the parameter defined in the procedure or function header.

For example, if you define a procedure with the header

```
procedure Strange(var r: univ MyRec; i: univ integer);
```

you could then call it with any of the following statements.

```
Strange(nil, 400);
Strange(100000, 'c');
Strange(@'mystring', true);
```

Forward and Extern

Occasionally, it is necessary to deal with procedures and functions that cannot be specified so that they are defined before use, or that are not actually defined in the Pascal program itself. These problems are overcome by the use of special directives.

The ISO Pascal standard requires that the `forward` directive be provided. When you declare a procedure or function as forward, you write the declaration the same way that you normally would, but the header and statement parts are replaced by `forward`. Later in the declaration part, the procedure or function must be declared again, this time with no parameter list or return type, but with a declaration part and a statement part.

```
procedure used (a: integer); forward;

procedure callit (a: integer);

begin
if a < 0 then used(a);
end;

procedure used;

begin
callit(-a);
end;
```

The `extern` directive is used when a procedure or function is defined outside of the Pascal program or unit. This occurs in two common cases:

1. A procedure or function is written in assembly language.
2. A library of common procedures or functions are available and called from several programs, and you decide to use the extern directive rather than using the interface file with a uses statement.

When any of these occur, the procedure or function declaration must be made at the program level. As with the `forward` directive, the `extern` directive replaces the header and statement parts.

```
procedure doit (x, y: integer); extern;
```

ISO Standard Pascal does not require the `extern` directive. While many compilers support it, its use must be considered to be nonstandard. ORCA/Pascal also allows the spelling external for compatibility purposes. While the shorter name is more common, a few programs being ported from other machines may use the longer name. ∆

Tool, UserTool, Vector and ProDOS

ISO The directives described in this section are extensions to Standard Pascal. ∆

ORCA/Pascal provides four other directives which are similar to `extern`, named `ProDOS`, `tool`, `UserTool`, and `vector`.

`ProDOS` is placed immediately after a procedure or function heading, and requires an integer-valued parameter, enclosed in parentheses. (The parenthesis are optional for compatibility with other Pascals, but should always be coded.) It is used to tell the compiler that the procedure or function just declared is a ProDOS 16 system call, a GS/OS system call, or an ORCA/Shell call, and the integer value is the number of the system call. The call numbers are given in the ProDOS Technical Reference Manual, Apple IIGS GS/OS Reference, and ORCA/M Reference Manual, respectively.

Example:

```
buffer: char;                    {call ProDOS to read from a file}
readDCB = record
   rdRef: integer;
   rdBuff: ^buffer;
   rdCount: longint;
   rdReq: longint
end;

procedure read (var dcb: readDCB); ProDOS(47);
```

`Tool` is placed immediately after a procedure or function heading, and requires two integer-valued parameters, enclosed in parentheses. (Again, the parenthesis are optional for compatibility, but should always be coded.) It is used to tell the compiler that the procedure or function just declared is a tool call. The first parameter is the tool number, and the second parameter is the number of the tool call. The tool and call numbers are given in the Apple IIGS Toolbox Reference manuals, volumes 1 through 3, and Programmer's Reference for System 6.0.

`UserTool` looks just like a `Tool` directive, but tells the compiler to generate a call to a user tool, rather than a system tool. User tools are discussed in the Apple IIGS Toolbox Reference manuals.

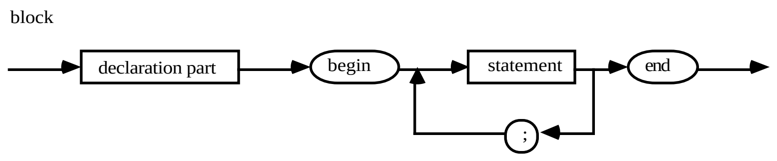
Example:

```
function IMVersion: versionNumber;  tool(11, 4);

{integer math tool call to determine version number of integer }
{math tool set                                                  }
```

Vector is used to make tool-like calls to programs that do not use the tool or user tool vectors. One example of such a program is HyperCard IIGS. This directive also uses two parameters enclosed in parenthesis. The first is a longint; this is the address of the vector to call. The second parameter is the tool number. The net effect of this directive is to tell the compiler to use toolbox conventions for passing parameters and returning function values; and to make the call by loading the X register with the tool number (the first parameter), then doing a JSL to the vector address (the first parameter). You can find examples of this directive in HyperXCMD.pas, which is the interface file for HyperCard IIGS.

`Tool`, `UserTool`, `Vector` and `ProDOS` calls are designed to be used with the interface files provided with ORCA/Pascal. These files are described in Chapter 4.

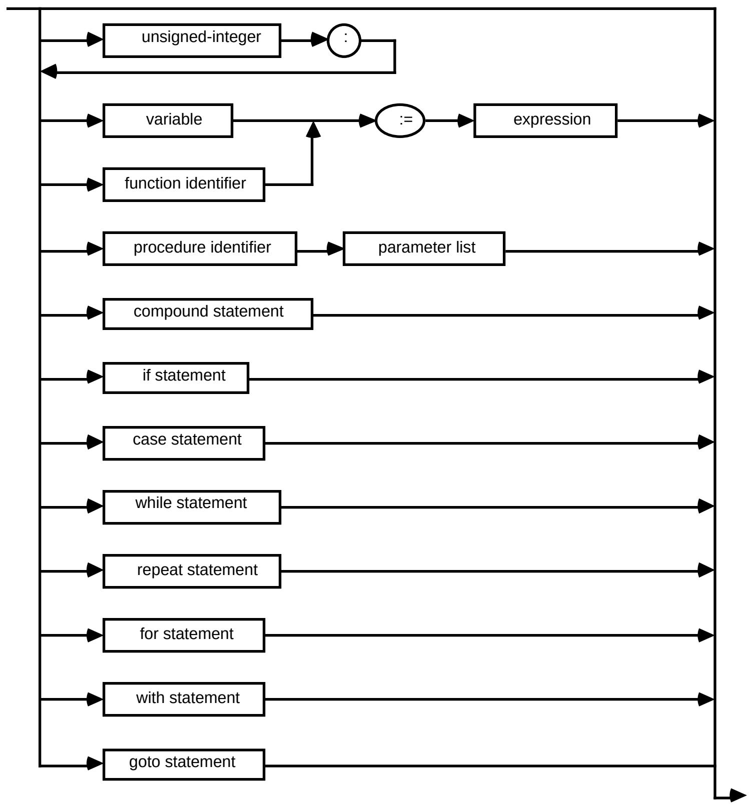
Chapter 19 - The Block



A block is the part of a procedure, function or program that contains local declarations and the statements to execute. It consists of a declaration part, the keyword `begin`, zero or more program statements separated by semicolons, and the keyword `end`. The statements are executed one after another until the end of the block is reached. When the end of the block is reached, all local declarations cease to exist and control returns to the calling program, procedure or function. If the block is the program block, control returns to the shell or program launcher.

Chapter 20 - Statements

statement



The Assignment Statement

The assignment statement allows a value to be assigned to a variable. The value can be a constant, another variable, or a more complicated expression. General rules for coding the expression part are covered in the next chapter. The variable can be a simple variable, a dynamic variable pointed to by a pointer, an array, an array element, a record, an element of a record, an object, or the name of a user-defined function.

The assignment statement is coded as a variable followed by the assignment operator and an expression. The value of the expression is calculated and the result replaces the value of the variable.

Pascal is a strongly typed language. The type of the expression must be assignment compatible with the type of the variable or the compiler will flag an error. An expression V2 is type compatible with a variable V1 if any of the following conditions are met:

1. They are the same type.
2. One is a subrange of the other, or they are both subranges of the same host type.
3. Their ordinal base types are compatible, and either both types are packed, or neither type is packed.
4. Both types are strings.
5. Both values are one of the real types; these include real, double, comp and extended.

ISO In Standard Pascal, strings are fixed length arrays of characters. If {$ISO+} has been specified, the strings must also have the same number of components. Δ

The second type of compatibility is assignment compatibility. V1 is assignment compatible with V2 if one of the conditions stated below is met. If V1 is assignment compatible with V2, then you can assign V2 to V1, as in V1 := V2.

1. V1 and V2 are the same type, but they are not file types and do not have components that are file types.
2. V1 is one of the real types (real, double, comp, extended) and V2 is integer or longint.
3. V1 and V2 are type compatible ordinal types, and the of value of V2 falls in the range of values valid for V1.
4. V1 and V2 are type compatible sets, and the members of the set V2 fall in the range of legal members for set V1.
5. V1 and V2 are type compatible string types.
6. V1 is longint, and V2 is integer.
7. If {$ISO+} has not been specified, then V1 can be a string, and V2 a character.
8. If {$ISO+} has not been specified, then V1 can be an object, and V2 can be an object of the same type, or of a type that is a descendent of the type of V1, or V2 can be nil.

Two results derived from this definition of assignment compatibility differ from some other common languages, and so deserve special mention. First, very little automatic type conversion takes place. In fact, the only time types are converted during assignment is when: the expression is an integer type and the variable is one of the real types (real, double, comp, extended); the expression is a longint type and the variable is integer; the expression is an integer type and the variable is longint. Secondly, Pascal allows the assignment of structured types. So long as two structured types are compatible, the assignment takes place as a single statement, often replacing a loop in other languages. For example, with the declaration

```
a,b: array[1..10, 1..10] of real;
```

the assignments

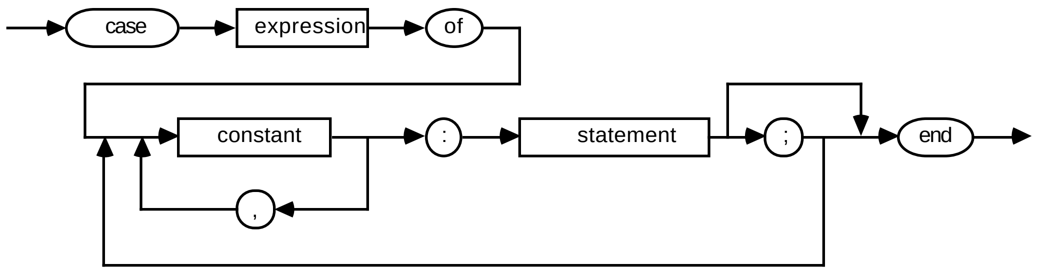
```
a := b;
a[3] := a[10];
```

are legal.
The type byte is a subrange of type integer; therefore values of type byte can be assigned to integer variables. An integer expression which falls within the range 0..255 can be assigned to a byte variable.
The type longint is an entirely new type. Expressions of type longint can be assigned to byte or integer variables, but only if the `ord` operation given below is applied first. Expressions of type byte or integer can be assigned to longint variables, however. To assign a longint value to an integer, you can use the following operation:

```
integerValue  :=  ord (longintExpression)
```

The types real, double, extended and comp are treated as the same type. Although the various formats use differing amounts of storage, all operations are performed using extended format numbers.
Assigning one object to another does not copy the contents of one object to a new spot, as it would for a record assignment. Instead, when you assign one object to another, both variables refer to the same object. See Chapter 14 for a complete discussion of object assignment.

Case Statement



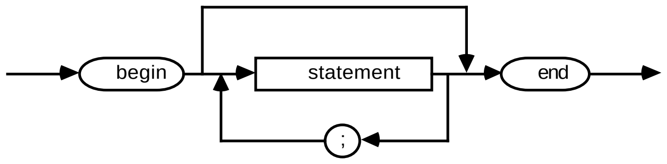
The case statement allows you to choose one item from a list of possible alternatives. The type of the expression must be compatible with the type of the case constants. The expression is first evaluated. Control is then passed to the statement after the corresponding case constant. It is an error if there is no corresponding case constant. You can avoid this error by using the otherwise clause. If none of the case constants matches the expression, the otherwise clause is executed. After the appropriate statement has been executed, the statement after the case statement is executed.
Some Pascal compilers do not allow the colon after the otherwise clause. We recommend using the colon, but it is actually optional in ORCA/Pascal.

ISO Using otherwise in a case statement is an extension to Standard Pascal. In Standard Pascal, it is illegal to execute a case statement when there is no case label that matches the case value, and the compiler is expected to stop with a run-time error if this happens. ORCA/Pascal will, in fact, stop with a run-time error if you execute a case statement with no matching case label and no otherwise clause. Δ

Example:

```
case i of
   1,3,5: writeln('odd');
   2,4,6: writeln('even');
   otherwise: writeln('neither')
   end;
```

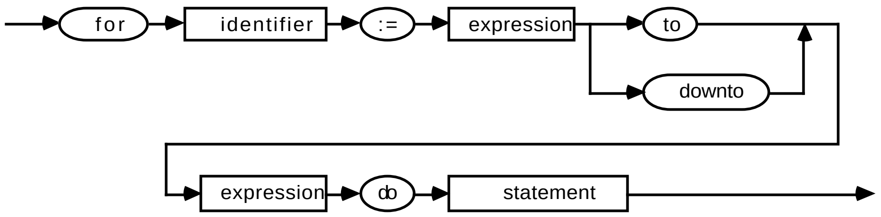
Compound Statements



There are many places in Pascal where it is useful to group a series of statements into a single syntactic unit. For example, you might want a for loop to include several statements instead of just one. The compound statement allows for this. Anywhere that a statement is legal in Pascal, a compound statement can be used. The compound statement takes the form of the keyword `begin`, a series of statements separated by semicolons, and the keyword `end`. The statements are then treated as a block, as illustrated in the following loop that initializes arrays.

```
for i := 1 to 10 do begin
   a[i] := 0;
   b[i] := maxint;
   end;
```

For Statement



The for loop allows for repetitive looping when the exact number of times to execute the loop can be computed before the loop starts. At execution time, the loop control variable is assigned the starting value specified by the expression. The second expression is then evaluated, and the result saved. Next, the looping process starts. Each loop begins by testing to see if the value of the loop variable is less than or equal to the termination value specified by the second expression (or greater than or equal for a downto loop). If it is, the statement is executed, the loop control variable is incremented (or decremented, for downto) and the process repeats.

It is possible for a loop to not be executed. This happens when the loop control variable starts out larger than the termination value.

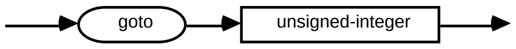
Pascal protects the loop variable by requiring that it not be placed in danger of being changed while the loop is in progress. This means that you cannot assign a value to the loop control variable while inside the loop. You cannot use the same loop control variable for two nested loops (although you can use the same variable on successive loops), and you cannot pass the loop control variable to a procedure or function as a var parameter. Pascal also requires that the loop control variable be defined locally.

The value of the loop variable is not defined after the loop terminates.

Some examples of for loops are:

```
for i := 0 to 100 do
   writeln(i);

for i := 1 to 10 do
   for j := 1 to 10 do
      matrix[i,j] := 0.0;

for time := 10 downto 0 do
   writeln
      ('T minus ',time:1,' seconds, and counting.');
```

Goto Statement



The goto statement allows control to be passed to another statement in the program. The reserved word `goto` is followed by a label number. The label must have been declared in the declaration section, and must appear in exactly one place in the block.

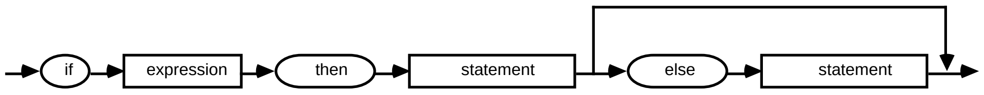
It is illegal to jump into a structure via a goto statement, such as into the middle of a for loop, a begin-end block, or the body of a case statement. It is illegal to branch from an if statement into its else clause with a goto. It is not illegal to branch out of a structure with a goto statement. For example:

```
for i := 1 to 10 do begin
   read(line[i]);
   if eoln then goto 1;
   end;
1:;
```

is legal, but you could not branch from outside the loop into the loop. It is also legal to branch from one procedure or function into its enclosing procedure or function.

```
procedure infinite_loop;

label 1;

   procedure strange;
   begin
   goto 1;
   end;

begin
1: strange;   {a very strange infinite loop}
end;
```

If-Then-Else Statement



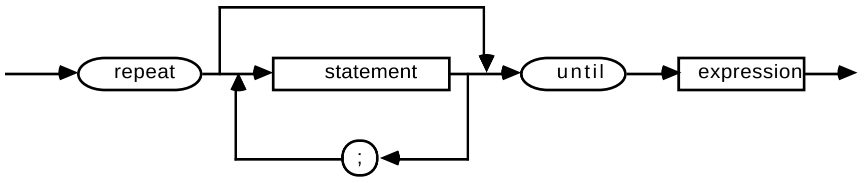
The if statement allows a statement to be executed only if a condition is met. It has two forms, one when an alternate statement should be executed if the condition is not met (an else is used) and one where nothing is done if the condition is not met. The expression must result in a boolean value. Examples of `if` statements are:

```
if a = 10 then
   a := 1
else
   a := a+1;

if today = Monday then
   let_it_rain;
```

Probably the most common error in coding an if statement is to place a semicolon before the else clause. Keep in mind that the semicolon separates statements, it does not terminate them. Finally, a useful construct is the nested if statement, which is often used when all values of a variable cannot be listed for a case statement.

```
if month in
   [January,March,May,July,August,October,December]
   then days := 31
else if month in [April,June,September,November] then
   days := 30
else begin
   if leapyear then
      days := 29
   else
      days := 28
   end;
```

Repeat Statement



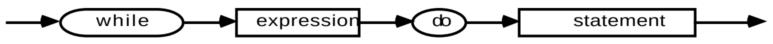
The repeat statement is used when a loop must be executed at least one time, but the number of times to loop cannot be computed when the loop starts. First, the body of the repeat statement is executed. The expression is then evaluated. The expression must be boolean. If its value is false, the loop is executed again; otherwise, the statement after the repeat statement is executed.

The repeat statement is rather unusual in Pascal, in that the body of the loop does not need to be a single statement. Like the compound statement, the loop body can be a series of statements separated by semicolons.

Examples:

```
repeat
   read(myfile,ch);
   process(ch)
until eof(myfile);

repeat
   getmove;
   makemove;
   update_board
until game_done;
```

While Statement

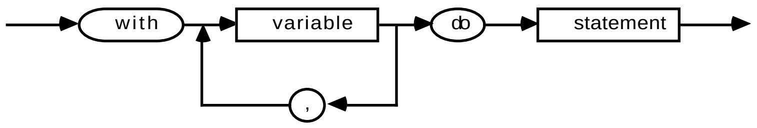


The while statement is used when the body of a loop may not need to be executed at all. The boolean expression is evaluated. If its result is true, the statement is executed and the process repeats. If it is false, the statement after the while statement is executed.

Examples:

```
while not eoln(myfile) do begin
   read(myfile, ch);
   process(ch);
   end;

{draw a circle}
a := delta;
MoveTo(mid_x + length, mid_y);
while a <= twopi do begin
   x := round(mid_x + cos(a)*length);
   y := round(mid_y + sin(a)*length);
   LineTo(x, y);
   a := a + delta;
   end;
```

With Statement



The with statement provides a shorthand method for accessing the fields within a record. The variables specified in the with statement are the names of variables of type record, and these names are implied to be prefixed to any field names occurring within the statement portion of the with. The with statement is applied to the variables in the order in which they occur. This rule is important when two or more variables have the same record type. For example, consider the following declarations and assignments:

```
type
   outfit = record
      shirtSize: integer;
      pantSize: integer
      end;

var
   man1, man2: outfit;

begin
with man1, man2 do
   shirtSize := 15;
   ...
```

The field `shirtSize` refers to the variable `man2`. To set the `shirtSize` field for `man1`, the following code would have to be used:

```
with man1, man2 do begin
   shirtSize := 15;
   man1.shirtSize := 15
   end;
```

The with statement example above could also be written as

```
with man1 do
   with man2 do
      shirtSize := 15;
```

This structure more clearly shows the scope rules applied to with variables, and thus how `shirtSize` is "local" to `man2`.

Chapter 21 - Expressions

Operators

There are a variety of operators which can be used to manipulate data. These are summarized in the table below. The functions they perform are discussed in sections dealing with the data types that the operators work on.

ISO With ORCA/Pascal, all operations defined for values of type real are also defined for values of type double, extended and comp. All operations defined for type integer are also defined for values of type byte and longint. Δ

Binary Operators	Types	Operation
+	integer, real, set	addition
-	integer, real, set	subtraction
*	integer, real, set	multiplication
/	integer, real	division
**	integer, real	exponentiation
div	integer	integer division
mod	integer	modulus arithmetic
in	scalar, set	test inclusion
=	integer, character, string, real, set	test equality
<>	integer, character, string, real, set	test inequality
<=	integer, character, string, real, set	test less than or equal
>=	integer, character, string, real, set	test greater than or equal
<	integer, character, string, real	test less than
>	integer, character, string, real	test greater than
and	boolean	logical and
or	boolean	logical or
&	integer	bitwise and
|	integer	bitwise or
!	integer	bitwise exclusive or
<<	integer	bit shift left
>>	integer	bit shift right

Unary Operators	Types	Operations
-	integer, real	negation
+	integer, real	ensure value is positive
not	boolean	logical not
@	any variable or structure	take address of
~	integer	bitwise not

Operations on Integers and Long Integers

The operators +, -, and * perform addition, subtraction, and multiplication, respectively, in the normal mathematical sense.

The / operator performs division in the normal mathematical sense. Both operands can be of type integer or longint, but the result will be real.

The ** operator raises the first operand to the power given in the second operand. Both operands can be of type integer or longint, but the result will be of type real. The first operand must be greater than zero.

The div operator performs integer division in the normal way - that is, the result is the same as performing a mathematical division and then truncating the fractional part of the result.

The mod operator returns the remainder resulting from the division of two integer quantities. The `mod` operator for `i mod j` is defined as the smallest positive number that can result from the expression `(i - (k * j))`, where `k` is also an integer. The examples below illustrate how these operators work.

Operation	Result
1 + 2	3
1 - 2	-1
10 * 20	200
5 / 2	2.5
2 ** 3	8.0
22 div 10	2
10 div 22	0
(-41) div 10	-4
10 mod 3	1
(-7) mod 5	2
9 mod 3	0

The following restrictions must be observed, or a run-time error will result.

1. The result of any operation involving at least one longint value must be in the range [-maxint4 - 1 .. maxint4]; otherwise, the result for only integer operands must be in the range [-maxint -1 .. maxint]. Maxint is defined as 32767; maxint4 is defined as 2147483647.
2. The second operand of the div operator must not be zero.
3. The second operand of the mod operator must be greater than zero.

Integer and longint values can also be compared. The comparison operators take two integer arguments and produce a boolean result. These examples illustrate the results produced by the comparison operators.

Operation	Result
1 < 2	true
2 < -3	false
2 < 2	false
1 > 2	false
2 > -3	true
2 > 2	false
100 >= 99	true
100 >= 100	true
-4 >= 5	false
100 <= 99	false
100 <= 100	true
-4 <= 5	true
14 = 14	true
-10 = 10	false
14 <> 14	false
-10 <> 10	true

All of the operators discussed so far, except div and mod, can also be used with real operands, or with a mixture of real and integer operands. See the section discussing operations on reals for details.

The bit operators work only with integer and long integer operands. These operators cannot generate run-time errors. The table below illustrates how they work:

Operation	Example	Result
and	4 & 3	0
	7 & 5	5
or	4 | 3	7
	7 | 5	7
eor	4 ! 3	7
	7 ! 5	2
shift left	3 << 2	12
	6 << 4	96
shift right	7 >> 2	1
	240 >> 4	15

All of the unary operators except not can be applied to integers. The + operator actually has no effect on the program. While there are certain places where it cannot be used due to the syntax of the Pascal language, in can always be replaced by a space without affecting the program produced by the compiler. The - operator negates the value. This has the effect of subtracting the argument from zero, so that

```
-i
```

will produce the same result as

```
0 - i
```

The ~ operator performs a bitwise negation of an integer. That is, in the binary representation of the number all zeroes are converted to ones, and all ones are converted to zeroes. For example,

```
~15 = -16
```

Operations on Reals

ISO All operations defined for real numbers are also valid for data of type double, extended or comp. ∆

The binary operators `+`, `-`, and `*` take two real arguments or one real and one integer argument and produce a real result. The `/` operator and the `**` operator take any combination of real or integer operands and produce a real result. If the result is not in the range of numbers that can be represented by a real number, a run-time error will occur.

These operators perform addition, subtraction, multiplication, division, and raise one number to the power of another in the normal mathematical sense. The examples below illustrate how the operators work.

Operation	Result
1.0 + 3.4	4.4
3.14 - 0.6	2.54
2.3 - 3.0	-0.7
1.0 / 0.4	2.5
1/2	0.5
3 * 4.1	12.3
3 ** 4	81.0
2 ** 0.5	1.414213

The operations are performed using a limited precision subset of the real numbers, as are all computations on a computer. This can lead to some unexpected results. For example, 1.0+1e-20 is not 1.00000000000000000001, as you might expect; it is still 1.0.

The following restrictions must be observed, or a run-time error will result.

1. The result of any operation must be in the subset of real numbers that can be represented.
2. The second operand of the `/` operator must not be zero.
3. The first operand of the exponentiation operator must be greater than zero.

Reals can also be compared to other reals, or to integers. The comparison operators take two real arguments, or one real argument and one integer argument, and produce a boolean result. When a real value is compared to an integer or longint value, the integer is first converted to a real, and then the comparison takes place as if the comparison was of two real arguments.

Two of the unary operators can be applied to reals. The `+` operator actually has no effect on the program. While there are certain places where it cannot be used due to the syntax of the Pascal language, in can always be replaced by a space without affecting the program produced by the compiler. The - operator negates the value. This has the effect of subtracting the argument from zero, so that

```
    -3.14
```

will produce the same result as

```
    0.0 - 3.14
```

Operations of Characters

The only operations valid on characters are comparisons. Testing for equality or inequality is straight forward. Testing for greater than or less than implies some ordering of the character set. The ordering used in ORCA/Pascal is the same as for the ASCII character set. In all cases, comparing two characters c1 and c2 will give the same result as comparing ord(c1) and ord(c2).

Standard Pascal requires that the digits be ordered sequentially. For example, ord('0')+1 must be equal to ord('1'). The standard requires that alphabetic characters be ordered, but it makes no requirement that they be sequential. Thus, 'A' must be less than 'B', but there is no requirement that ord('A')+1 be equal to ord('B'). The ISO standard does not specify the ordering of ordinal values of uppercase and lowercase letters.

ORCA/Pascal on the Apple IIGS uses the ASCII character set, so alphabetic characters are both ordered and sequential. Uppercase characters are less than lowercase characters, so that 'A' < 'a'.

Operations on Strings

The only operations valid on strings are comparisons. The comparison is performed character by character until a difference occurs. Thus, 'apple' is greater than 'able,' but less that 'ax.'

If two strings are different in length but identical up to the last character in the shorter string, the shorter string is less than the longer one. For example, 'apple' is less than 'apples.'

ISO If the {$ISO+} directive has not been specified, strings of different lengths can be compared, and strings can be compared to characters. In both cases, the comparison functions as if the shorter structure is extended to the length of the longer structure with characters whose ordinal values are zero. Δ

Operations on Booleans

There are two binary boolean operators, both giving boolean results. And gives a result of true if both arguments are true, and a result of false if either argument is false. Or gives a result of true if either argument is true, and a result of false if both arguments are false.

Not is the only unary operator that takes a boolean argument. It gives true if the argument is false, and false if the argument is true.

Operations	Result
false and false	false
false and true	false
true and false	false
true and true	true
false or false	false
false or true	true
true or false	true
true or true	true
not true	false
not false	true

Operations on Sets

Three binary operators take set arguments and yield set results. The + operator performs a set union. The - operator gives the difference between two sets. The * operation gives the intersection of two sets.

Operation	Result
[1,2,4] + [1,3,4]	[1,2,3,4]
[1,2,4] - [1,3,4]	[2]
[1,2,4] * [1,3,4]	[1,4]

One binary operator, in, takes a scalar for the first argument, a set for the second, and produces a boolean result. It tests to see if the scalar is a member of the set.

Operation	Result
1 in [1,2,4]	true
3 in [1,2,4]	false

Sets can be tested for equality and inequality. Two sets are equal if they have the same members. Sets can also be tested to see if all of the members of one set are also members of the other set. `S1 <= s2` is true if all members of the set `s1` appear in the set `s2`. `S1 >= s2` is true if all members of `s2` are in `s1`. The < and > comparisons are not defined for sets.

Operation	Result
[1,2,4] = [1,3,4]	false
[1,2,4] = [1,2,4]	true
[1,2,4] <> [1,3,4]	true
[1,2,4] <> [1,3,4]	false
[1,2,4] <= [1,2,3,4]	true
[1,2,3,4] <= [1,2,4]	false
[1,2,4] <= [1,2,4]	true
[1,2,4] >= [1,2,3,4]	false
[1,2,3,4] >= [1,2,4]	true
[1,2,4] >= [1,2,4]	true

Operations on Pointers

ISO

Using @ as an operator to extract an address is an extension to Standard Pascal. In Standard Pascal, @ is used as an alternate character for ^, which is not available in some character sets. In ORCA/Pascal, unless the ISO directive has been used to enforce strict compliance with the ISO standard, the @ operator can be used for either purpose. The compiler picks between the two uses based on context. ∆

The @ operator returns the address of a data object. It can be used to obtain the address of a procedure, function, variable, record, array, record element, array element, or string constant. The value returned is a pointer that is type-compatible with nil; that is, it is assignment-compatible with any pointer.

The example below shows how pointer arithmetic can be performed to randomly access the elements of an array. Other than addition and subtraction of integers, it is not recommended that you apply mathematical functions to pointers. For instance, the addition of two pointers is somewhat meaningless; the square root of a pointer value is truly nonsense.

```
procedure PointerMath (var num: array [0..99] of integer);

var
   p: ^integer;
   x, i: integer;

begin
p := @num;
x := 1;
offset := 5;
```

```
{ Loop to place a value in every 5th position of the array num. }
for i := 0 to 19 do begin
   p^ := x;
   x := x * 3;
   p := pointer ( ord4(p) + (2 * offset) )
end;
```

The `ord4` operation turns `p` into a longint so that arithmetic may be performed. Adding `2 * offset` to `p` moves `p` forward in the array to access the next integer (each integer is two bytes long) that is `offset` elements away.

Operator Precedence

Operator precedence is what causes 1 + 2 * 3 to be 7 instead of 9. In Pascal, expressions that have several operators in a row, with each operator of equal precedence, are evaluated from left to right. For example, the integer math operation 100 div 3*2 gives 66 if the div operation is performed first, and 16 if the multiplication comes first. In Pascal, the div is performed first.

The operators are shown below, with the highest precedence shown first. Operators with the same precedence are shown on the same line.

```
not     ~       **      @
*       /       &       <<      >>      div     mod     and
+       -       |       !       or
=       <=      >=      <       >       <>      in
```

The order in which the operands of a binary operator are evaluated is implementation dependent. For example, consider the function `changeit`:

```
function changeit(var x: integer): integer;

begin
changeit := x div 2;
x := x div 3;
end;
```

Now consider how it is used in this expression:

```
x * changeit(x)
```

If `x` is 5, and the left term is evaluated first, then the value of the expression is 10. If, however, the function is called first, the value of the expression is 2. The ISO standard says that the order of evaluation is implementation dependent, which means that each implementation of Pascal can choose the order in which the terms are evaluated. In ORCA/Pascal, the left term is always evaluated first. If you plan to move your programs to other compilers, you should not write expressions that depend on the implementation-defined order of evaluation, like the one above.

Type Compatibility Rules

There are two kinds of type compatibility rules in Pascal. The first defines when two types are compatible. When two types are compatible, they may be substituted for one another as parameters to procedures and functions, and used as operands to the various operators described above. If they are not type compatible, and you attempt to use them this way, an error will be flagged. For two types to be compatible, they must satisfy one of the following conditions:

1. They are the same type.
2. One is a subrange of the other, or they are both subranges of the same host type.
3. Their ordinal base types are compatible, and either both types are packed, or neither type is packed.
4. Both types are strings.
5. Both values are one of the real types; these include real, double, comp and extended.

ISO Standard Pascal does not support variable length strings. If {$ISO+} has been specified, the strings must have the same number of components. Δ

The second type of compatibility is assignment compatibility. V1 is assignment compatible with V2 if one of the conditions stated below is met. If V1 is assignment compatible with V2, then you can assign V2 to V1, as in V1 := V2.

1. V1 and V2 are the same type, but they are not file types and do not have components that are file types.
2. V1 is real and V2 is integer.
3. V1 and V2 are type compatible ordinal types, and the of value of V2 falls in the range of values valid for V1.
4. V1 and V2 are type compatible sets, and the members of the set V2 fall in the range of legal members for set V1.
5. V1 and V2 are type-compatible string types.
6. V1 is longint, and V2 is integer.
7. If {$ISO+} has not been specified, then V1 can be a string, and V2 a character.
8. If {$ISO+} has not been specified, then V1 can be an object, and V2 can be an object of the same type, or of a type that is a descendent of the type of V1, or V2 can be nil.

Under ORCA/Pascal, an operation defined for integers will accept operands which are any combination of types integer, longint, or byte. If one of the operands is a longint value, then the result will also be of type longint.

Any scalar value can be coerced into an integer by the operation:

```
ord (scalar)
```

If the scalar value is of type longint, you will need to check if the value is within the range -maxint to maxint before performing the coercion to avoid causing a run-time error. The operation ord(pointer) will always fail unless the address is in bank zero.

An integer may be converted to a character value using the function chr, which takes a single integer argument, and returns a character result.

Any scalar value can be coerced into a longint by the operation:

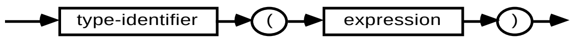
```
ord4 (scalar)
```

This operation is especially useful for converting a pointer into a longint so that pointer arithmetic may be performed, as in

```
ord4 (pointer)
```

The types real, double, comp and extended are treated as the same type. All of these types are converted to extended before calculations are performed, then converted back to the proper format for storing.

Type Casting



ISO Type casting is an extension to Standard Pascal. It is not available if the ISO directive has been used to enforce strict compliance with the ISO Pascal standard. Δ

Type casting lets you change the type of any scalar or pointer to any other scalar or pointer. The result can appear anywhere that a variable or expression of that value may be used. Type casting tells the compiler to treat one value as if it is of a different type. Unlike the built-in functions that convert from one type to another in Pascal, such as ord and chr, type casting does not convert from one value to another.

▲ **Warning** The fact that type casting is not a conversion operation has very important implications. For example, if you cast a longint value to an int, and pass the result as a parameter, you have told the compiler to push four bytes onto the run-time stack, but to treat it as if only two bytes were pushed.

Type casting should never be used when a type conversion function exists to do the same job. Type casting exists in Pascal for pointer type conversions and for tricks with overlaying records. Using type casting routinely when a data conversion function should be used can cause unexpected results or crashes. ▲

For example, let's assume that you have defined a record as follows:

```
long = record
   lsw,msw: integer;
   end;
```

Then, using type casting, you could extract the most significant word of a long integer by

```
var
   x: integer;
       l: longint;

begin
x := long(l).msw;
```

In this example, specifying `long` tells the compiler to treat the value as a record with the format of the record long. Once the type is converted this way, you can extract either word of the four byte argument. Another common use of type casting is to convert an integer into an enumerated type. For example:

```
type
   color = (red, blue, green);

var
   c: color;

begin
c := color(1);
```

assigns the color blue to the variable c.

While type casting is an advanced concept, it can be very useful. In the first example, assuming long is a global variable, the code generated for the entire construct would be the single machine language instruction, `LDA LONG+2`.

Chapter 22 - Built-in Procedures and Functions

This chapter covers the procedures and functions that are built into the compiler, and do not need to be declared as extern to be available. They are not all required by the ISO Pascal standard. When the functions or procedures are not a part of Standard Pascal, this fact is noted in the subroutine description by marking it with an ISO bullet. Use of a nonstandard procedure or function, along with the {$ISO+} directive in your program, will cause the extended procedure/function to be flagged as an error.

Table 22.1 lists the procedures and functions in functional groups. The descriptions themselves are alphabetized for easy reference.

The memory management calls refer to the heap. The heap is the area of memory that can be used for variables. The heap is used in two distinct ways: as a stack frame area and for dynamic variables. Stack frames are the memory set aside for local variables that must be allocated each time a procedure or function is called. Dynamic variables are variables allocated by the program via calls to the procedure new.

As dynamic variables are allocated and deallocated, unused space can appear between the currently allocated dynamic variables. This space is reused when new variables are allocated. There is no garbage collection as such.

Heap Management	Use
dispose	return allocated memory to free memory pool
new	allocate memory from free memory pool

Input and Output	Use
close	close a file
eof	true if at end of file
eoln	true if at end of line
get	primitive read
open	open a file for both input and output
page	write an ASCII form-feed character to an output file or standard output
put	primitive write
read, readln	read from file or standard input
reset	open a file for input
rewrite	open a file for output
seek	go to random position in a file
write, writeln	write to a file, standard output, or standard error output

Mathematics	Use
abs	absolute value of argument
arccos	arc cosine of argument
arcsin	arc sine of argument
arctan	arc tangent of argument

arctan2	arc tangent of two arguments
cos	cosine of argument
exp	exponent of argument
ln	natural logarithm of argument
odd	true if argument is an odd number
random	returns pseudo-random real number
RandomDouble	returns pseudo-random double-precision real number
RandomInteger	returns pseudo-random integer
RandomLongint	returns pseudo-random four-byte integer
round	round argument to nearest whole number (returns an integer)
round4	round argument to nearest whole number (returns a longint)
seed	initialize random number generator
sin	sine of argument
sqr	square of argument
sqrt	square root of argument
tan	tangent of argument
trunc	truncate fractional part of real number (returns an integer)
trunc4	truncate fractional part of real number (returns a longint)

String Manipulation	Use
concat	concatenate two strings
copy	extract substring from string
delete	remove substring from string
insert	insert a string within another string
length	current length of a string, in characters
pos	position in string where substring begins

Miscellaneous	Use
chr	convert integer to character equivalent
CommandLine	returns command line entered when program was executed
cnvis	convert integer to string
cnvrs	convert real to string
cnvsd	convert string to double-precision real value
cnvsi	convert string to integer
cnvsl	convert string to longint value
cnvsr	convert string to single-precision real value
EndDesk	shut down the desktop environment
EndGraph	shut down the graphics environment
halt	abort program by issuing a ProDOS QUIT call
member	see if an object is a member of an object family
ord	convert character to integer equivalent
ord4	convert ordinal value to longint equivalent
pack	move values from unpacked array into packed array
pointer	convert pointer to untyped pointer

pred	value in collating sequence of ordinal object which precedes argument object
seed	initialize the random number generator
ShellID	identifier of shell under which program was executed
SizeOf	returns the size of a variable or type
StartDesk	initialize desktop environment
StartGraph	initialize graphics environment
succ	value in collating sequence of ordinal object which succeeds argument object
SystemError	trap run-time errors
ToolError	number of error generated by last tool call made
unpack	move values from packed array into unpacked array
UserID	identification number assigned to program by loader

Table 22.1 Summary of the Built-in Procedures and Functions

Abs

```
function abs (x: real): real;
```

The absolute value function takes a single argument. It returns the absolute value of the argument. The argument must be assignment-compatible with the types real, longint, or integer. The result type is extended if the argument was real, double or comp; integer if the argument was integer; and longint if the argument was of type longint.

Example:

```
a := abs(a);
```

ArcCos

```
function arccos (x: real): real;
```

ISO ArcCos is an extension to Standard Pascal. Δ

The arccos function returns the arc cosine of the argument. The argument must be assignment-compatible with a real value. The result is expressed in radians, and will be in the range 0 .. π. The result is an extended value.

Example:

```
angle := arccos(z);
```

ArcSin

```
function arcsin (x: real): real;
```

ISO ArcSin is an extension to Standard Pascal. Δ

The arcsin function returns the arc sine of the argument. The argument must be assignment-compatible with a real number. The result is expressed in radians, and will be in the range -π/2 .. π/2. The result is an extended value.

Example:

```
angle := arcsin(z);
```

ArcTan

```
function arctan (x: real): real;
```

The arctan function returns the arc tangent of the argument. The result is expressed in radians, and will be in the range -π/2 .. π/2. The argument must be assignment-compatible with a real value, and must be expressed in radians. The result is an extended value.

It is an error to take the arctan(0).

See the library function arctan2, below, for an alternate version of this function.

Example:

```
    angle := arctan(x/y);
```

ArcTan2

```
function arctan2 (x, y: real): real;
```

ISO ArcTan2 is an extension to Standard Pascal. Δ

The arctan2 function returns the arc tangent of two arguments. The first argument is the x coordinate of a point in a Cartesian coordinate system, and the second argument is the y coordinate of the point. The arguments must be assignment-compatible with a real value. The result is expressed in radians, and will be in the range -π .. π; it represents the angle between the positive x-axis and the point. The result is an extended value.

Chr

```
function chr (x: integer): char;
```

The chr function converts an integer into a character. The integer must be in the valid range for a character, which is 0 to 127.

Example:

```
function upper(ch: char): char;

{return an uppercase character}

begin
if ch in ['a'..'z'] then
   ch := chr(ord(ch) - ord('a') + ord('A'));
upper := ch;
end;
```

Close

```
procedure close (f: fileVariable);
```

ISO Close is an extension to Standard Pascal. Δ

Close is a nonstandard procedure used to close a file. It is not necessary to close a file under most conditions, since the compiler will close it automatically when execution of the block in which the file was declared is complete. The only time you really need to close a file is if you need to use a ProDOS function that requires the file be closed before it will work. The close call accepts one parameter, the file variable assigned to the opened file.

Example:

```
close(f);
```

CommandLine

```
procedure CommandLine (var cmndLine: string[size]);
```

ISO CommandLine is an extension to Standard Pascal. Δ

The CommandLine procedure accepts a var parameter, which is a string that will hold the command line passed to the program when the program was executed. The command line includes the name of the program and any parameters, with I/O redirection stripped and handled by the shell. The size of the string must be large enough to hold the characters typed. The maximum size of the command line is 255 characters. If no command line was passed, the string will be null.

Example:

```
procedure handleParms;

var
   cmndLine: string[255];

begin
CommandLine (cmndLine[255])
{ Put code to handle parameters here }
end;
```

Cnvds

```
function cnvds (x: extended; width, digits:integer; ):
string[size];
```

ISO Cnvds is an extension to Standard Pascal. Δ

Returns the character string equivalent of the real argument. If the string is not large enough to hold the floating-point representation then truncation occurs at the end of the string. If the string will be formatted in exponential notation, then the string should be at least nine characters long. The second parameter specifies the field width for the number to be written in the string. If it is zero, then the string is to be written in exponential format. The third parameter gives the number of digits to written after the decimal point.

Example:

```
x := 3.5e3;
str := cnvds (x, 5, 1);
```

would set str to the characters 3500.0.

Cnvis

```
function cnvis (i: integer): string[size];
```

ISO Cnvis is an extension to Standard Pascal. Δ

Returns the character string equivalent of the integer argument. If the string is not large enough to hold the converted value, then truncation occurs at the end of the string. The string should be at least six characters long.

Example:

```
var
   str: string[2];
   number: string[5];

begin
str :=  cnvis (157);
number := cnvis (157);
```

would set str to '15' and number to '157.'

Cnvrs

```
function cnvrs (x: real; width, digits: integer): string[size];
```

ISO Cnvrs is an extension to Standard Pascal. Δ

Returns the character string equivalent of the real argument. If the string is not large enough to hold the floating-point representation then truncation occurs at the end of the string. If the string will be formatted in exponential format, the string should be at least nine characters long. The second parameter specifies the field width of the number to be written in the string. If it is zero, then the string is to be written in exponential format. The third parameter specifies the number of digits to be written after the decimal point.

Example:

```
x := 3.5e3;
str := cnvrs (x, 6,1);
```

would set str to the characters 3500.0.

Cnvsd

```
function cnvsd (s: string): extended;
```

ISO Cnvsd is an extension to Standard Pascal. Δ

Cnvsd returns real number represented by the input string argument. A string too large to be represented in an extended value will be flagged as an error.

Example:

```
    z := cnvsd ('34.4e5');
```

Cnvsi

```
function cnvsi (s: string): integer;
```

ISO Cnvsi is an extension to Standard Pascal. Δ

Returns the integer represented by the input argument string. If the string is too large to be represented in an integer, an error will be flagged.

Example:

```
    k := cnvsi ('36');
```

Cnvsl

```
function cnvsl (s: string): longint;
```

ISO Cnvsl is an extension to Standard Pascal. Δ

Returns the long integer representation of the string. If the string is too large to be represented in a longint value, then an error is flagged.

Example:

```
    xl := cnvsl ('1234567');
```

Cnvsr

```
function cnvsr (s: string): real;
```

ISO Cnvsr is an extension to Standard Pascal. Δ

Returns the single-precision floating-point representation of the input string argument. If the string is too large to be represented in a real value, an error is flagged.

Example:

```
x := cnvsr ('333.5');
```

Concat

```
function concat (str1: string[size1]; str2: string[size2]; ... ):
                 rstr[rsize];
```

ISO Concat is an extension to Standard Pascal. Δ

The concat function accepts two or more strings as parameters, and returns a string as a result. The first string is placed into the beginning of the resulting string, the second string is appended to the end of the first string, the third string is appended to the end of the second string, and so on. Thus, the string that is returned is the concatenation of all of the string parameters. The length of the new string is the combined length of all of the parameters. If the string that will be receiving the result of the concatenation is not large enough, truncation will occur at the end of the string. For example:

```
var
   s1, s2: string[5];
   s3: string[7];

begin
s1 := 'pas';
s2 := 'cal';
s3 := concat (s1, s2);
```

would result in s3 being set to pascal. If, however, s3 had been defined as

```
s3: string[4];
```

the concat operation would result in s3 being set to `pasc`.

Copy

```
function copy (str: string[size]; index, count: integer):
   string[rsize];
```

● ISO Copy is an extension to Standard Pascal. Δ

The copy function accepts three parameters and returns a string. The first parameter is a string from which a substring is to be copied into the resulting string. The second parameter is an integer, and denotes the character in the string where copying is to begin. The third parameter is an integer, and gives the number of characters to copy. If *index* exceeds the length of *str*, then the null string is returned. If the resulting string is not large enough to hold the substring, then truncation occurs from the end of the string.

Example:

```
var
   str1: string[5];
   str2: string[10];
   str3: string[3];

begin
str1 := 'apple';
str2 := copy (str1, 2, 3);
str3 := copy (str1, 1, 4);
```

would result in str2 being set to ppl and in str3 being set to app.

Cos

```
function cos (x: real): real;
```

The cos function returns the cosine of the argument, expressed in radians. The argument must be assignment-compatible with a real variable. The result is an extended value.

Example:

```
function secant(r: real): real;

begin
secant := 1.0/cos(r);
end;
```

Delete

```
procedure delete (var str: string[size]; index, count: integer);
```

ISO Delete is an extension to Standard Pascal. Δ

The delete procedure accepts three parameters. The first is a string from which characters are to be deleted. The second is an integer which specifies the character at which deletion is to begin. The third is an integer which gives the number of characters to be removed. If *index* exceeds the length of *str*, the string is left intact. If *index+count* is greater than the length of the string, then characters are removed from *index* to the end of the string.

Example:

```
   var str: string[10];

   begin
   str := 'upside ';
   delete (str, 4, 10);
```

would result in str being set to ups.

Dispose

```
procedure dispose (var p: pointer;
   v1, v2, ..., vn: variantConstants);
```

The dispose procedure releases the memory allocated to a pointer variable or object, removing it from the heap and freeing the heap space for future use by other dynamic variables. The first parameter is a pointer to the area to be disposed. The remaining parameters, if specified, are constants from a variant record declaration. Standard Pascal requires that this list of variant record constants match the list that was used when the dynamic variable was allocated with new, but does not require compilers to verify that the restriction is met. ORCA/Pascal does not check to make sure that the lists match.

Examples:

```
   dispose(ptr);
   dispose(ptr, reccons, recconst2);
```

EndDesk

```
procedure EndDesk;
```

ISO EndDesk is an extension to Standard Pascal. Δ

Shuts down any tools started by the StartDesk procedure, described later in this section, and returns the program to the text environment.

EndGraph

```
procedure EndGraph;
```

ISO EndGraph is an extension to Standard Pascal. Δ

Shuts down any tools started by the StartGraph procedure, described later in this section, and returns the program to the text environment.

Eof

```
function eof (f: fileVariable): boolean;
```

Eof tests a file variable to see if the last value read from the file was the last one in the file. The parameter, which is optional, is a file variable. If the parameter is left off, eof tests to see if an end-of-file character has been encountered from standard input. Under ORCA/Pascal, if standard input is defined as the keyboard, then the end-of-file character is the two-key sequence control@ . All other characters read from the keyboard cause eof to return false. If the standard input has been redirected to a file, eof will report an end-of-file condition as for an opened file.

Examples:

```
while not eof do begin
   getchar;
   processchar;
   end;

while not eof(myfile) do begin
   getchar;
   processchar;
   end;
```

Eoln

```
function eoln (f: fileVariable): boolean;
```

The eoln function takes an optional file variable as a parameter and returns a boolean result. The file variable must be a text file. If the parameter is left off, the file is assumed to be the standard input. The result is true if the last character read was the last character on the line, and false if it was not. Testing for eoln and getting a true result does not mean that the end-of-line character has already been read - it means that the next character in the file is the end-of-line marker. When you read the end-of-line mark, the character returned is always a space.

It is an error to call eoln for a file in which eof is true.

The following sample program illustrates the use of eoln and eof by reading a file from disk and listing it on the screen.

```
program list (output);

var
   ch: char;
   f: text;

begin
reset(f,'myfile');
while not eof(f) do begin
   if eoln(f) then begin
      readln(f);
      writeln;
      end
   else begin
      read(f, ch);
      write(ch);
      end;
   end;
end.
```

When the file is the standard input, and the input is coming from the keyboard, things work a little differently. In that case, lazy I/O is used. Lazy I/O provides a way for a Pascal program to read characters from an interactive device and give them to the program immediately. With normal file input, Pascal must read one character ahead to see if the next character is an end-of-file or end-of- line mark. Lazy I/O gets around this by reading the character and giving it to the program right away. Lazy I/O sets `eof` true when CTRL@ is entered from the keyboard; eoln is true when the return key is pressed. The character returned for the end-of-line mark will still be a space.

Exp

```
function exp (x: real): real;
```

The exp function returns the exponent of the argument. The argument must be assignment-compatible with a real value. The result is an extended value.

Example:

```
    function power (x, y: real): real;

    { raise x to the y power                                        }

    begin {power}
    power := exp(y*ln(x))
    end; {power}
```

Get

```
procedure get (f: fileVariable);
```

The get procedure takes a file variable as a parameter. It places the current file element into the internal read buffer, and then advances the file by one file element, so that the file pointer points to the next file element. It is an error to call get on a file that has reached the last element - i.e., for a file for which eof is true.

Example:

```
    get(f);
```

Halt

```
procedure halt (errorNumber: integer);
```

ISO Halt is an extension to Standard Pascal. Δ

The halt procedure is used to abort execution of a program and/or report a run-time error to the shell. Internally, halt issues a ProDOS QUIT call. If halt is used to simply exit a program early, the *errorNumber* parameter should be set to zero to inform the shell that no error occurred. If halt is used to exit upon detection of a serious error, *errorNumber* should be set to the error number as returned by a shell, ProDOS, or tool call. If the error is internal to the program, set *errorNumber* to -1. The shell uses the error number to determine if it needs to take some action, such as exiting an EXEC file. If portability is an issue, avoid use of halt.

Example:

```
if x < maxint then
   z := x * 2;
else
   halt(-1);
```

Insert

```
procedure insert(str1: string[size1]; var str2: string[size2];
   index: integer);
```

ISO Insert is an extension to Standard Pascal. Δ

The insert procedure allows *str1* to be inserted into *str2*, starting at *index*. If the insertion causes *str2* to overflow, the extra characters are lost at the end of the string.

Example:

```
var
   str1, str2: string[20];

begin
str2 := 'up and away';
str1 := ', up';
insert (str1, str2, 3);
```

would result in str2 being set to up, up and away.

Length

```
function length (str: string[size]): integer;
```

ISO Length is an extension to Standard Pascal. Δ

Length returns the current length of the string. If a null character has been embedded in a c-string, the length of the string is the number of characters up to, but not including, the null character. If null has not been written to the string, then the length of the string is the maximum size given when the string was defined. For length strings, the length is always the same as ord(str[0]), even if the string has imbedded null characters. If portability is an issue, avoid the use of length.

Example:

```
var
   x: integer;
   str: string[10];

begin
str := 'hey';
x := length (str);
```

would set x to 3.

Ln

```
function ln (x: real): real;
```

The ln function returns the natural logarithm of the argument. The argument must be assignment-compatible with a real value. The result is an extended value. It is an error if the argument is less than or equal to zero. See exp for an example.

Member

```
function member (obj: objectType; otype: objectType): boolean;
```

ISO Member is an extension to Standard Pascal. Δ

Member checks to see if an object is a member of an object family. The first parameter is an object variable; this is the object that will be checked. The second parameter is an object type. Member returns true if the object is the same type as the object type, or if the object is a descendent of the object type. (The object is a descendent of the object type of the type of the object is formed by inheriting the object type, or if there is an inheritance chain back to the object type.) If the object variable is nil, or if it is not a descendent of the object type, Member returns false.

Example:

```
type
   obj1 = object
      end;
   obj2 = object (obj1)
      end;

var
   v1: obj1;
   v2: obj2;

begin
new(v1);
new(v2);
writeln(Member(v1, obj2)); {prints false}
writeln(Member(v2, obj1)); {prints true}
```

New

```
procedure new (var p: pointer; v1, v2, ..., vn: variantConstant);
```

New is used to allocate dynamic variables and objects from the heap. One parameter is required; it is either a pointer or an object.

If the required parameter is a pointer, other parameters may be given after the pointer, which refer to the variant portions of a record if the pointer is to contain the address of a variant record. The type of the pointer determines how much space new will allocate from the heap. New returns a pointer to a heap area. It is an error if there isn't enough room on the heap for the allocation. The area returned is not initialized in any way.

Further control over the amount of memory that is allocated is available with the optional additional parameters. Normally, new will allocate the maximum amount of space that can ever be needed by a variant record. For example,

```
type
   data = (str, int);
   variant = record
      case kind: data of
         str: (string80: string[80]);
         int: (i: integer);
   end;

var
   p: ^variant;
```

defines a pointer p that can point to an integer, which requires two bytes of storage, or a string, which requires eighty-one. If p will point to either kind of data, it should be allocated like this:

```
new (p);
```

This application of new to a variant record allows for allocation of the maximum amount of memory that will ever be needed (83 bytes in our example – 81 bytes for the string plus two bytes for the variant selector, data). If, however, you know that once it is allocated, p will always point to an integer, you may use the call

```
new (p, int);
```

In this case, new would only allocate four bytes from the heap, two for the integer and two for the variant selector, data. If you do this, however, be sure that you do not store a string in p^.str later. Doing so will damage the heap, with unpredictable results.

If the required parameter is an object, new allocates space for a new object and initializes the space so the methods associated with the object type will be used when messages are sent to the object. Fields in the object are not initialized in any way.

Odd

```
function odd (k: integer): boolean;
```

The odd function takes an integer argument and returns a boolean result. The result is true if the argument is odd, and false if it is even.

```
procedure BinOut (i: integer);

{ recursively write an integer as a binary number                 }
{                                                                 }
{ parameters:                                                     }
{    i - integer to write                                         }

   procedure BitOut (i, c: integer);

   { write the bits                                               }
   {                                                              }
   { parameters:                                                  }
   {    i - integer containing bits                               }
   {    c - number of bits left to write                          }

   begin {BitOut}
   if c > 1 then
      BitOut(i div 2, c - 1);
   write(chr(ord('0') + ord(odd(i))))
   end; {BitOut}
```

```
begin {BinOut}
BitOut(i, 16)
end; {BinOut}
```

Open

```
procedure open (f: fileVariable, s: string);
```

ISO Open is an extension to Standard Pascal. Δ

Open is a nonstandard procedure that opens a file for both input and output. The first parameter is a file variable. An optional second parameter can be coded. It is a string which contains the path name of the file. The path name ends with the first space, null character (ASCII zero), or end of the string, whichever comes first. That is, a file name of 's ' would cause the file named S in the current prefix to be opened. If you are programming under the ORCA or APW shells, the string can also contain device numbers, prefix numbers, and .. , or .printer or .console. Note that .printer can only be used with output and erroroutput, and that .console can only be used with input, output, and erroroutput. Input, output, and erroroutput are specified in the program declaration part of the program.

If no file name is given in the open call, open will assign a unique file name. That file name will start with the characters "SYSPAS," and will have a four-digit number appended to "SYSPAS." The number will increment for each new file variable that is opened with reset, open or rewrite, and that does not have a programmer-assigned file name. For example, the first file created without an assigned file name will be given the name "SYSPAS0001." The file will be created on the work prefix, prefix 3.

The file pointer points to the first element of the file. If there was already information in the file, all old information remains. If there was no file by the given name, one is created and opened. Along with seek and the Standard Pascal I/O functions, open gives full access to files for both input and output, and allows random access to the elements of the file.

Example:

```
var
   myfile: file of stuff;
   f: file of integer;

open(myfile);
open(f, '/mydata/data1');
```

Ord

```
function ord (x: ordType): integer;
```

The ord function converts any ordinal type into an integer. The ordinal types include integers, enumerations, characters, and booleans. For enumerations, the value will be the number you get when you count the enumeration constants, starting from zero. For example, for the definition

```
    color = (red,orange,yellow,green,blue,violet);
```

the ordinal values of each of the enumeration constants are:

red	0
orange	1
yellow	2
green	3
blue	4
violet	5

The ordinal values for characters are given by the ASCII character set. For boolean values, ord(true) = 1 and ord(false) = 0.

Ord can be used to force the conversion of a long integer into an integer. This is occasionally necessary if you want to use a long integer value in a place where they are normally not allowed, such as for subscripting arrays.

See Type Casting in Chapter 21 for ways to convert from an integer to another ordinal type.

See chr for an example.

Ord4

```
function ord4 (x: ordType): longint;
```

ISO Ord4 is an extension to Standard Pascal. Δ

The functionality of ord4 is identical to the Standard Pascal function ord, except that it returns a long (four-byte) integer instead of a two-byte integer. Ord4 is especially useful for converting any pointer into an integer so that pointer arithmetic may be employed. If portability is an issue, avoid use of ord4.

Example:

```
var
   nums: array [1..maxint] of integer;
   p, q: ^integer;
   x: integer;

begin
p := @nums;
q := pointer (ord4(p) + 2);
x := q^;
```

would set x equal to the second element of the array nums.

Pack

```
procedure pack (UnpackedArray: array[n..m] of someType;
   start: integer;
   var PackedArray: packed array [n1..m1] of someType);
```

The pack procedure moves values from an unpacked array into a packed array. Let Packed_Array be a packed array, and Unpacked_Array be an unpacked array, with both arrays having elements of the same type. We will also need a starting subscript, which we will call Start. The starting subscript must be assignment-compatible with the index type of the unpacked array. With these assumptions, the call to pack

```
pack(Unpacked_Array, Start, Packed_Array);
```

will move components from the unpacked array into the packed array. The first component moved will be the component with subscript Start. Components are placed in the packed array starting with the first element in the packed array and continuing until the packed array is full. It is an error if there are not enough components in the unpacked array to fill the packed array.

Another way of looking at all this is to examine a piece of code that does the same thing as the pack procedure. With the declarations

```
Unpacked_Array: array[1..100] of components;
Packed_Array: packed array[Lower..Upper] of components;
Start: integer;
```

the call to pack shown earlier is completely equivalent to this code:

```
begin
j := Start;
for i := Lower to Upper do begin
   Packed_Array[i] := Unpacked_Array[j];
   if i <> Upper then j := j+1;
   end; {for}
end;
```

It is easy to see several points about packed arrays from this code. First, the packed and unpacked arrays do not have to be the same size. The also do not have to have the same type of index variable. Finally, it is easier to see why the unpacked array must have enough components to fill the packed array.

While the pack procedure is equivalent to the code shown above, it is considerably faster to use pack that it is to use the code shown.

Page

```
procedure page (f: fileVariable);
```

The page procedure writes an ASCII form-feed character (the ordinal value is 12) to the output file specified by the parameter. If no parameter is given, standard output is assumed.

The action that results from writing a form-feed character depends on the device that it goes to. If the device is the CRT screen, and if the screen has been initialized for Pascal output (which it has, if you are running under the ORCA text environment) then the screen is cleared and the cursor is placed at the top left corner of the screen. If the device is a printer, most printers will perform a page eject. The next character printed will be at the top of a new page. If the output is going to a disk file, a form-feed character is written to the file.

Example:

```
page (myfile);
```

Pointer

```
function pointer (x: pointerType): anonymousPointer;
```

ISO Pointer is an extension to Standard Pascal. Δ

The pointer function bypasses type-checking for pointers. Its parameter is any pointer, integer, or longint. It returns a pointer whose type is equivalent to the predefined constant nil. That is, it is type-compatible with any pointer. This procedure does not result in any generated code unless the argument is of type integer, in which case the two-byte integer is extended to four bytes. Avoid the use of pointer if portability is an issue. See ord4 for an example.

Pos

```
function pos (target: string[tsize]; source: string[ssize]):
   integer;
```

ISO Pos is an extension to Standard Pascal. ∆

The function pos searches the *source* string for the first occurrence of the *target* string. If *target* is found, it returns the number of the character where *target* begins. If *target* is not found, it returns zero.

Example:

```
   str := 'big program';
   x := pos ('gram', str);
```

would set x to 8.

Pred

```
function pred (x: ordinalType): ordinalType;
```

The pred function returns the value before the argument. The argument can be any ordinal value, and the result type matches the argument type. Ordinal values include integers, characters, enumerations, and booleans. It is an error if the value before the argument does not exist - for example, pred(false) is an error, since there is no value before false. Pred(true), however, is false.

Example:

```
   newcolor := pred(oldcolor);
```

Put

```
procedure put (f: fileVariable);
```

The value pointed to by the file pointer is written to the output file. It is an error to call `put` for a file that has reached end-of-file or that was opened for input.

Example:

```
program count;
{writes integers to an integer output file}

var
   f: file of integer;
   i: integer;

begin
rewrite(f, 'numbers');
for i := 1 to 100 do begin
   f^ := i;
   put(f);
   end;
end.
```

Random

```
function random: real;
```

ISO Random is an extension to Standard Pascal. Δ

The random function returns a pseudo-random real number in the range -1.0 .. 1.0, exclusive. The seed procedure, discussed later in this chapter, can be used to initialize the sequence of numbers generated.

Example:

```
procedure Simulation(i: integer);

var
   time: real;

begin {Simulation}
seed(i);
time := random;
beginSim (time)
end; {Simulation}
```

RandomDouble

```
function RandomDouble: extended;
```

ISO RandomDouble is an extension to Standard Pascal. ∆

The RandomDouble function returns a pseudo-random real number in the range -1.0 .. 1.0, exclusive. The seed procedure, discussed later in this chapter, can be used to initialize the sequence of numbers generated.

Example:

```
procedure Simulation (i: integer);

var
   time: real;

begin {Simulation}
seed(i);
time := RandomDouble;
beginSim (time)
end; {Simulation}
```

RandomInteger

```
function RandomInteger: integer;
```

ISO RandomInteger is an extension to Standard Pascal. ∆

The RandomInteger function returns a pseudo-random integer in the range -maxint .. maxint, exclusive. The seed procedure, discussed later in this chapter, can be used to initialize the sequence of numbers generated.

Example:

```
procedure Simulation (i: integer);

var
   people: integer;
```

```
begin {Simulation}
seed(i);
people:= RandomInteger;
beginSim (people)
end; {Simulation}
```

RandomLongint

```
function RandomLongint: longint;
```

ISO RandomLongint is an extension to Standard Pascal. Δ

The RandomLongint function returns a pseudo-random longint number in the range -maxint4 .. maxint4, exclusive. The seed procedure, discussed later in this section, can be used to initialize the sequence of numbers generated.

Example:

```
procedure Simulation (i: integer);

var
   population: longint;

begin {Simulation}
seed(i);
population := RandomLongint;
beginSim (population)
end; {Simulation}
```

Read, Readln

```
procedure read ( [f: fileVariable;] d1, d2, ..., dn: fileData);
procedure readln ( [f: fileVariable;] d1, d2, ..., dn: fileData);
```

Read is the easiest way to read from an input file. Read can take several parameters - the first can be a file variable. If the first parameter is not a file variable, it is assumed that the input will come from the standard input file, input. The remaining parameters are variables that are assignment-compatible with the type of the file. Values are read from the file and placed sequentially into the specified variables. Thus, read is really just a shorthand way of doing gets, and letting the compiler worry about assignments to the file buffer. For example, with a file defined as

```
myfile: file of stuff;
```

and variables

```
v1, v2, v3: stuff;
```

the call

```
    read (myfile,v1,v2,v3);
```

is completely equivalent to

```
v1 := myfile^;
get (myfile);
v2 := myfile^;
get (myfile);
v3 := myfile^;
get (myfile);
```

ISO In Standard Pascal, the read statement cannot be used to read strings from a text file. ∆

Read has some special abilities when the file is a text file. (The standard file input is one such file.) In that case, the variable can be a character, integer, real or string. If the variable is a character, one character is read and assigned to the variable. If the variable is an integer or real, read starts by skipping all spaces, end-of-line markers, and any control characters until it gets to a printing ASCII character. It is an error if that character is not the start of a number. Read then reads in the number, which stops with the first character that is not a part of the number, and assigns the value to the variable. For a string, read will read all characters up to the end of the line, or up to the length of the string, whichever is shorter. If the line was shorter than the string, and the string is a standard Pascal string, the remaining string characters are set to chr(0). If the string has a length byte, the length byte is set to the length of the string.

For example, consider a file produced by the following program:

```
program MakeFile;

{create a file of integers}

var
   i, j: integer;
   f: text;

begin
rewrite(f,'intfile');
```

```
for i := 1 to 10 do begin
   for j := 1 to 10 do
      write(f,(i-1)*10+j:10);
   writeln(f);
   end;
end.
```

When the file is read, there is no need to keep track of where the end-of-line markers are. This program reads the file created by the last example program and writes each integer on a separate line:

```
program writeint (output);

{writes the file created by the last program}

var
   f: text;
   i: integer;

begin
reset(f,'intfile');
while not eof(f) do begin
   if eoln(f) then
      readln(f)
   else begin
      read(f, i);
      writeln(i);
      end;
   end;
end.
```

Readln functions exactly like read, except that after the read is finished, characters are skipped until eoln becomes true, then the end-of-line marker itself is skipped. Readln can be coded with only a file variable, or with no parameters at all. If only a file variable is coded, characters are skipped until the end-of-line has been passed. If no parameters are coded, the same is done for the standard input.

Reset

```
procedure reset (f: fileVariable; s: string);
```

ISO The use of a string to set the file name is an extension to Standard Pascal. This extension is common in microcomputer based Pascals, but mainframe implementations of Pascal generally provide some other way to associate a file variable with a name. ∆

Reset opens a file for input. The first parameter is a file variable. An optional second parameter is optional. It is a string which contains the path name of the file. The path name ends with the first space, null character (ASCII zero), or end of the string, whichever comes first. That is, a file name of 's ' would cause the file named S in the current prefix to be opened. If you are programming under the ORCA or APW shells, the string can also contain device numbers, prefix numbers, and .. , or .console. Note that .console can only be used with the file input, which is specified in the program declaration part of the program.

If no file name is given in the reset call, ORCA/Pascal will open the file last associated with the file variable given as the first parameter to reset. This file can be one named by the programmer, or it can be a system-provided default name. The default name starts with "SYSPAS" and has a four-digit number appended to "SYSPAS." The number is incremented with each call to open, reset, or rewrite that does not use an assigned file name. See the open and rewrite procedures in this section for more information about associating an external file name with a file variable.

It is an error if the file does not exist. The file pointer is assigned the first value from the file. Resetting the standard input is legal, but has no effect.

Examples:

```
    reset(f);
    reset(g, '/mydata/data4');
```

ReWrite

```
procedure rewrite (f: fileVariable; s: string);
```

ISO The use of a string to set the file name is an extension to Standard Pascal. This extension is common in microcomputer based Pascals, but mainframe implementations of Pascal generally provide some other way to associate a file variable with a name. ∆

Rewrite opens a file for output. The first parameter is a file variable. An optional second parameter is optional. It is a string which contains the path name of the file. The path name ends with the first space, null character (ASCII zero), or end of the string, whichever comes first. That is, a file name of 's ' would cause the file named S in the current prefix to be opened. If you are programming under the ORCA or APW shells, the string can also contain device numbers, prefix numbers, and .. , or .console or .printer. Note that .console and .printer can only be used with the files output and erroroutput, which are specified in the program declaration part of the program.

If the file exists, all of its old contents are first deleted. The first put will write a value to the first position in the file. Rewriting the standard output is allowed, but is ignored.

If no file name is given in the rewrite call, rewrite will assign a unique file name. That file name will start with the characters "SYSPAS," and will have a four-digit number appended to "SYSPAS." The number will increment for each new file variable that is open or rewritten, and that does not have a programmer-assigned file name. For example, the first file created without an assigned file name will be given the name "SYSPAS0001." The file will be created on the work prefix, prefix 3.

The type of the file depends on the type of the file variable. If the file variable was declared as text, the file created will be a ProDOS TXT file. If the file variable was declared as anything else, even file of char, the output file will be a ProDOS BIN file with an aux field of zero.

Example:

```
    rewrite(f);
    rewrite(g, '../myoutput/outfile1');
```

Round

```
function round (x: real): integer;
```

The argument is converted to the nearest integer. If the nearest integer is outside the range -maxint to maxint, an error results. The argument can be real, double, comp, extended, integer, or longint.

Example:

```
    round(2.5) = 3
    round(-2.5) = -3
    round(17.3) = 17
```

Round4

```
function round4 (x: real): longint;
```

ISO Round4 is an extension to Standard Pascal. ∆

The argument is converted to the nearest longint. If the nearest longint is outside the range -maxint4 to maxint4, an error results. The argument can be real, double, comp, extended, integer, or longint.

Example:

```
round4(2.5) = 3
round4(-2.5) = -3
round4(100000.3) = 100000
```

Seek

```
procedure seek (f: fileVariable; fptr: longint);
```

ISO Seek is an extension to Standard Pascal. ∆

The seek procedure provides a way to position the file pointer. This allows random access I/O. Seek takes two parameters - a file variable and an integer. The next value written to or read from the file will be the ith element, counting from zero.

△ **Important** Seek positions the file pointer, but does not do a get. To read a specific value from a file, you must first do a seek, then do a get, and then dereference the value. Using read statements, you would use seek to position the file pointer, then use read to fetch the value, then use read a second time to actually read the value. △

Example:

```
seek(f,pos+10);
```

Seed

```
procedure seed (x: integer);
```

ISO Seed is an extension to Standard Pascal. Δ

The seed procedure is used to initialize the random number generator. For any given seed, the random number generator will generate the same sequence of numbers. For example, if you use the seed 4 each time you run a program, you will get exactly the same results. This is handy when you are debugging a program. Once the program is complete, you should use some semi-random source to initialize the random number generator. One such source is the system clock; for an example of this, see the Artillery program in Chapter 4.

ShellID

```
procedure ShellID (var id: string[8]);
```

ISO ShellID is an extension to Standard Pascal. Δ

Each shell running on the Apple IIGS computer has a unique eight-character identifier. The shell identifier for ORCA and APW is BYTEWRKS. The procedure ShellID returns the string assigned to the shell that a program was executed under. If portability is an issue, avoid use of ShellID.

Example:

```
var
   str: string[8];

begin
ShellID(str);
```

Sin

```
function sin (x: real): real;
```

The sin function returns the sine of the argument, which is expressed in radians. The argument must be assignment-compatible with a real value. The result is an extended value.

Example:

```
function cosecant(r: real): real;

begin
cosecant := 1.0/sin(r)
end;
```

SizeOf

```
function SizeOf (value): integer;
```

ISO SizeOf is an extension to Standard Pascal. Δ

The SizeOf function returns the size, in bytes, of any type or variable. The parameter is a type name or a variable name. If the parameter is a type, SizeOf returns the number of bytes that would be required to hold a variable of the given type.

Example:

```
size := SizeOf(myRecord);
```

Sqr

```
function sqr (x: real): real;
function sqr (k: integer): integer;
```

The result is the square of the argument. The result is an extended value. It is an error is the result is too large to be represented. See sqrt for an example.

Sqrt

```
function sqrt (x: real): real;
```

The sqrt function returns the square root of the argument. The result is an extended value. The argument can be real, double, comp, extended, integer, or longint. The argument must be positive or zero, or an error will result.

Example:

```
function len (x, y: real): real;

{ find the length of a vector                                       }

begin {Len}
len := sqrt(sqr(x) + sqr(y))
end; {Len}
```

StartDesk

```
procedure StartDesk (x: integer);
```

ISO StartDesk is an extension to Standard Pascal. ∆

The StartDesk procedure is included with ORCA/Pascal to help simplify the task of writing desktop applications. The procedure accepts a single integer parameter of either 320 or 640, the graphics mode in which the application is to run. StartDesk initializes these tools:

Miscellaneous Tools	QuickDraw II	Desk Manager
Event Manager	Integer Math Tool Set	Window Manager
Menu Manager	Control Manager	QuickDraw II Auxiliary
LineEdit Tool Set	Dialog Manager	Scrap Manager
Standard File Tool Set	Font Manager	List Manager
Resource Manager		

SANE and the Memory Manager are started in all C programs, so they are not started explicitly by StartDesk.

The procedure EndDesk will shut these tools down.

StartDesk also opens your program's resource fork, if there is one, and makes it the current resource fork. The resource fork is opened with read access.

After using StartDesk, Pascal's standard input and output subroutines (like readln and writeln) draw characters to the graphics screen using the current font and pen position.

If your application needs any other tools, you are responsible for initializing and shutting them down.

Example:

```
StartDesk(320);
```

StartGraph

```
procedure StartGraph (x: integer);
```

ISO StartGraph is an extension to Standard Pascal. ∆

The procedure StartGraph is provided with ORCA/Pascal to make writing graphics applications easier. It accepts a single integer parameter of either 320 or 640, the graphics mode desired for the application. StartGraph initializes QuickDrawII, clears the graphics screen and then sets it to black, sets the pen color to white and puts the pen in "or" mode, and sets the foreground color to white and the background color to black. An example of a graphics program is given in Chapter 4.

After using StartGraph, Pascal's standard input and output subroutines (like readln and writeln) draw characters to the graphics screen using the current font and pen position.

Example:

```
StartGraph (640);
```

Succ

```
function succ (x: ordinalType): ordinalType;
```

The succ function returns the value after the argument. The argument can be any ordinal value, and the result type matches the argument type. Ordinal values include integers, characters, enumerations, and booleans. It is an error if the value after the argument does not exist - for example, succ(true) is an error, since there is no value after true. Succ(false), however, is true.

Example:

```
{write the printing ASCII characters}
ch := ' ';
while ch <= '~' do begin
   write(ch);
   ch := succ(ch);
   end; {while}
```

SystemError

```
procedure SystemError (errorNumber: integer);
```

ISO SystemError is an extension to Standard Pascal. Δ

When a run-time error occurs, the program can call a procedure named SystemError to handle it. By placing this procedure in your program, you can prevent your program from stopping with a terminal error. Once you have handled the error, you can exit the procedure where the error occurred, whereupon execution will pick up where it left off, or you can call another procedure in your program. The errorNumber corresponds to the number of the error, as given in the table below.

You can flag the error in either of two ways. First, you can print your own error message (or print no message at all). Second, you can call a system procedure called SystemPrintError with the error number. This procedure will print a text error message to error out and return to you.

If you decide to stop program execution, this can be done in one of two ways. First, you can use the halt procedure to stop, returning an error code to the shell if you wish. Second, you could call the system procedure SystemErrorLocation, which will print the line number and procedure or function name where the error occurred, and a trace back showing what calls were made. If you have turned off line numbers and procedure names using the names compiler directive, the call will produce no output. SystemErrorLocation causes your program to halt execution.

Error Number	Error
1	Subrange exceeded
2	File not open
3	Read while at end of file
4	I/O error
5	Out of memory
6	EOLN while at end of file
7	Set overflow
8	Jump to undefined case statement label *This error cannot be recovered from!*
9	Integer math error
10	Real math error
11	Underflow
12	Overflow
13	Divide by zero
14	Inexact
15	Stack overflow
16	Stack error

Table 22.2 Run-time errors

These three procedures – SystemError, SystemErrorLocation and SystemPrintError – are not predefined in Pascal, like the other procedures defined in this chapter. If you want to call one of these procedures from your program, you must define the procedure as external. To replace the procedure that is normally used, place the replacement procedure in your program just like you would any other globally accessible procedure.

For a detailed explanation of these run-time errors, see "Execution Errors" in Appendix B.

Tan

```
function tan (x: real): real;
```

ISO Tan is an extension to Standard Pascal. ∆

The tan function returns the tangent of the argument, expressed in radians. The argument must be assignment-compatible with a real value. The result is an extended value. It is an error to take the tangent of an angle which is a multiple of π/2. Avoid the use of tan if portability is an issue.

Example:

```
z := tan(x/y);
```

ToolError

```
function ToolError: integer;
```

ISO ToolError is an extension to Standard Pascal. ∆

ToolError returns the error number returned by the last call to a tool, ProDOS, or the shell, or zero if no error occurred. If a tool call produced an error, but then a subsequent tool call did not result in an error, a call to ToolError would yield a zero. Note also that many of the tool calls do not report errors of any kind. If portability is an issue, avoid use of ToolError.

Example:

```
k := ToolError;
if k <> 0 then
   halt(k);
```

Trunc

```
function trunc (x: real): integer;
```

Truncates the real argument, returning an integer result. The result is the largest integer that is less than or equal to the argument for positive arguments, and the smallest integer greater than or equal to the argument for negative arguments. The result must be in the valid range for integers or an error will result.

Examples:

```
trunc(1.9) = 1
trunc(-1.9) = -1
trunc(0) = 0
```

Trunc4

```
function trunc4 (x: real): integer;
```

ISO Trunc4 is an extension to Standard Pascal. Δ

Truncates the real argument, returning a longint result. The result is the largest long integer that is less than or equal to the argument for positive arguments, and the smallest long integer greater than or equal to the argument for negative arguments. The result must be in the valid range for long integers or an error will result.

Examples:

```
trunc4(100000.9) = 100000
trunc4(-1.9) = -1
trunc4(0) = 0
```

Unpack

```
procedure unpack (packedArray: packed array [n..m] of someType;
         unpackedArray: array [n..m] of someType; start: integer);
```

The unpack procedure moves values from a packed array into an unpacked array. Let Packed_Array be a packed array, and Unpacked_Array be an unpacked array, with both arrays having elements of the same type. We will also need a starting subscript, which we will call Start. The starting subscript must be assignment compatible with the index type of the unpacked array. With these assumptions, the call to unpack

```
unpack(Packed_Array,Unpacked_Array,Start);
```

will move all of the components of the packed array into the unpacked array. The first component moved will replace the component of the unpacked array whose subscript is `Start`. Components are placed in the unpacked array starting with the first element in the packed array and continuing until the packed array is empty. It is an error if there are not enough components in the unpacked array to accept all of the components of the packed array.

Another way of looking at all this is to examine a piece of code that does the same thing as the unpack procedure. With the declarations

```
Unpacked_Array: array[1..100] of components;
Packed_Array:  packed array[Lower..Upper] of components;
Start: integer;
```

the call to unpack shown earlier is completely equivalent to this code:

```
begin
j := Start;
for i := Lower to Upper do begin
   Unpacked_Array[j] := Packed_Array[i];
   if i <> Upper then j := j+1;
   end; {for}
end;
```

It is easy to see several points about packed arrays from this code. First, the packed and unpacked arrays do not have to be the same size. They also do not have to have the same type of index variable. Finally, it is easier to see why the unpacked array must have enough components to accept all of the packed array.

While the unpack procedure is equivalent to the code shown above, it is considerably faster to use unpack that it is to use the code shown.

UserID

```
function UserID: integer;
```

ISO UserID is an extension to Standard Pascal. Δ

The function UserID returns the number assigned to your program by the loader (a part of ProDOS 16) when it loaded your program. The UserID is required for some tool calls, most notably the memory manager. You should always allocate memory using the user ID returned by this function. This allows the shell to deallocate memory allocated by your program if your program terminates early.

The UserID returned by this function is the user ID for your program ored with $0100. This allows you to easily dispose of all memory allocated since your program started. The new procedure uses this user ID also, but you should not use the toolbox's memory manager to deallocate all memory for your user ID if you are using new.

Example:

```
x := UserID;
```

Write, Writeln

```
procedure write ( [f: fileVariable;] d1, d2, ..., dn: fileData);
procedure writeln ( [f: fileVariable;] d1, d2, ..., dn: fileData);
```

Write gives a shorthand way to write values to a file. It takes a variable number of parameters. The first can be a file variable - if so, the values are written to the indicated file. If the first parameter is not a file variable, the standard output file output is assumed. Each of the remaining parameters, of which there must be at least one, must be assignment-compatible with the data pointed to by the file pointer. The data is written to the file in the order given. Thus, with the declarations

```
myfile: file of stuff;

v1, v2: stuff;
```

the call

```
write (myfile, v1, v2);
```

is completely equivalent to

```
myfile^ := v1;
put (myfile);
myfile^ := v2;
put (myfile);
```

Writeln works just like a write, except that, after the values have all been written, an end-of-line marker is written. ORCA/Pascal writes the ASCII RETURN character (ordinal value 13) to mark the end of a line. Writeln is only valid with text files. It is legal to omit the value parameters with writeln, or to omit all of the parameters. Thus,

```
writeln;
```

writes a carriage return to the standard output.

When used with text files, write has some special properties. First, the types of the parameters can be mixed. They can include characters, strings, integers, reals, and booleans. Each of these can optionally be followed by a colon and a positive integer that gives the width of the field in which the value will be written. If the field width is larger than the number of characters needed to represent the value, spaces are written to right-justify the value in the field. If no field width is specified, a default size is used. The default varies with the type of the output value.

Type	Default Field Width
char	1
boolean	8
byte	8
integer	8
longint	16
real	16
double	16
extended	16
comp	16
string	length of the string

Table 22.3 Default output field widths

If the field width is too small to write the complete value, and the value to be written is not a string or boolean, the field width is ignored and all characters are written. If the output value is a string, characters are removed from the end of the string until it fits in the field. This is an internal operation – a string variable is not modified by a write operation.

Characters written to the screen are those from the ASCII character set, exactly as expected. The same is true for strings.

A boolean value of true is written as the string 'true.' Like string output, if the field is too small, the end characters are truncated, so write(true:1) would write the character 't' to the file. A boolean value of false is written as the string 'false,' with truncation applied as for true boolean values.

Integers are written with a leading minus sign if the value is negative, followed by the character representation for the integer value. No leading zeros are written. Long integers are written in the same way as integers.

If the field width of a real, double, comp or extended value is not specified, the number is written in scientific notation, using the default field width. If the number is negative, it starts with a minus sign. If the number is positive, the first character in the field is blank. This is followed by a digit, a decimal point, seven more digits, and an exponent. The exponent is written as an 'e', followed by a signed three-digit exponent value. For example:

```
 3.1415930e+000
-1.0000000e-047
```

The smallest field width possible for one of the real types is nine, since the format required for scientific notation is

```
+/-d.de+/-ddd
```

If a field width is given which is less than nine, the field width is ignored and the default format is written. If the field width is larger than nine, digits are written after the decimal point to bring the field size to the width specified. The largest field width possible under ORCA/Pascal is limited by 80 columns. That is, if the number begins at the start of a new line, then the maximum field size is exactly 80. You should also be aware that real numbers contain a maximum of seven significant digits. For field widths larger than 16, the extra digits after the seventh digit are inexact. Double values contain a maximum of 15 significant digits. For field widths larger than 22, the extra digits after the fifteenth digit are inexact. Extended values contain a maximum of 19 significant digits.

Real, double, comp and extended values can have another width after the field width. Like the field width, it is coded as a colon and a positive integer. The integer specifies the number of fractional digits, which is the number of digits to the right of the decimal place. When this parameter is coded, the number is no longer written in scientific notation. This might be used to write dollar amounts, as in

```
write('$',yourmoney:1:2);
```

which would print a suitably large number to the screen (We hope!):

```
    $12593.16
```

Examples:

```
write (f,v1,v2,v3);
writeln ('Hello, world.');
writeln (output,'Hello, world.');       {same effect as writeln above}
write (f,10,20,30,30+10);
write ('c','string',true,10,10.0);
write ('c':1,'string':6,true:8,10:8,10.0:16);
   {same effect as write above}
write ('c',string,true:4,10:2,10.0:4:1);       {all run together}
write (10.0:1:2);
```

Chapter 23 - Compiler Directives

Compiler directives are used to control the output of the compiler. They are coded much like a comment, with the opening token being {$ or (*$. No spaces are allowed between the opening comment character, { or (* , and the dollar sign character, $. What follows the name of the directive depends on which directive is used. Any directive can be followed by a comma and another directive, except for append and copy. Some directives must appear before the beginning of the program, while others are allowed to appear anywhere in the source file.

ISO While most compilers will provide some method of doing the things these directive do, the ISO standard does not require these directives. For that reason, the directives will need to be changed if you port the program to another compiler. ∆

Append

The append directive is followed by a string. The string contains the path name of a source file. The path name can contain device numbers, prefix numbers, and **..** . If the path name contains only a file name, then the current prefix is assumed. The current source file is closed and processing continues with the specified source file. Any characters after the append in the original source file are ignored.

Examples:

```
{$append 'myfile'}
{$append '/mypascal/project.x/file1'}
{$append '../file53'}
(*$append '.d4/prog4.pas' *)
```

CDev

The CDev directive tells the compiler to create a control panel device (CDev) rather than a normal program. CDevs are programs called under the control of Apple's Control Panel NDA. The directive must appear before the program statement to have any effect. The program must still have a body, but there should be no statements in the program body, and it is not executed.

The directive itself has a single parameter, the name of a function that the Control Panel will call.

For a short sample CDev, see the CDEV.SAMPLES folder of your ORCA/Pascal samples disk. For a complete technical description of CDevs and how to write them, see Apple II File Type Notes for file type $C7.

Example:

```
    {$ cdev main }                    { Create a CDev }
```

ClassicDesk

The ClassicDesk directive tells ORCA/Pascal that your program is a classic desk accessory, and that the compiler needs to generate some special code. This directive must appear before the program token. The directive has three parameters. The first is a string; this is the name that will be displayed in the classic desk accessory menu. The next two are the names of two procedures, the startup procedure and the shutdown procedure. The format is:

```
    {$ClassicDesk 'name' Start ShutDown}
```

Refer to Chapter 4 or the Apple IIGS Toolbox Reference Manual:Volume 1 for more information about writing classic desk accessories.

Copy

The copy directive is followed by a string. The string contains the path name of a source file. The path name can contain device numbers, prefix numbers, and **..** . If the path name contains only a file name, the current prefix is assumed. The compiler checks to make sure that the next token is a *) or }, then closes the current source file and opens the one specified. Once the end of the new source file has been reached, the old source file is reopened, and the compiler continues from the character after the end of the comment token containing the copy directive.

Copied files can copy other files. The level of nesting is limited only by available memory.

Examples:

```
    {$copy '/mypascal/project/file99'}
    (*$copy '5/prog3' *)
```

DataBank

Pascal assumes that the data bank register is correct - that is, that it points to the bank where the global variables are located. There are occasions where a Pascal procedure or function will be called by some other language, and this assumption may not be a good one. The most common case is when a procedure or function is called from an Apple IIGS tool. This directive tells the compiler to generate code at the start of each procedure or function to set the data bank to the global variable bank upon entering the procedure or function. The original data bank value is restored before returning to the caller.

Using this directive does not prevent a procedure or function from being called directly from Pascal. It does, however, increase the size of code and decrease execution speed a little. See the ToolParms directive for an example of the DataBank directive.

Debug

The debug directive is used to turn source level debugging on and off from inside the program. The directive currently supports two flags. If bit 0 is set (a value of 1), the compiler generates debug code and symbol tables for the source-level debugger. If bit 1 is set (a value of 2), the compiler generates only the debug information used by the profiler, so the program runs at a speed that is closer to true speed. You can use both flags together.

This directive may be expended in the future, If so, it will be expended in such a way that {$debug -1} gives all available debugging, while {$debug 0} turns off all debug code.

While the directive can be used anywhere in a program, debug code should be turned on and off only between program level procedures and functions. There are a few tricks you can do by turning debug code off and pack on in the middle of a subroutine, but be sure each subroutine ends with the same debug options in effect that were in effect when the subroutine started.

See the description of the ASML command in Chapter 8 for a way to turn debug code on and off from the shell or from a script file. See the description of the Compile... dialog in Chapter 7 for a way to turn debug code on and off from the PRIZM desktop development environment.

Example:

```
{$debug -1}
```

Dynamic

The dynamic directive looks, and for the most part works, like the segment directive, described later in this chapter. The only difference between the two is that static segments (like those created with the segment directive) are loaded when the program starts to execute, while dynamic segments are only loaded if one of the procedures or functions in the dynamic segment is called. In addition, by using Apple's loader (described in Apple IIGS GS/OS Reference), you can unload dynamic segments to free memory for other uses.

See the description of the segment directive for a general discussion of segments.

Eject

If the compiler listing has been redirected to a file or to the printer, then the eject directive causes a form-feed character (ASCII 12) to be written to the compiler listing. There are no operands for this directive. This directive has no effect if output is going to the console.

Examples:

```
{$eject}
(*$eject *)
```

Float

The 68881 floating-point card is supported under the 2.0 ORCA languages via an alternate SysFloat library. There are two required steps and one optional step involved in using the FPE card.

1. You must replace the SysFloat library in your libraries folder with the FPE version of this library. The FPE version of the library is located on the Extras disk at the path :Extras:FPE:SysFloat. This file should be moved to your libraries folder, replacing the file 13:SysFloat.

2. You must define and call a procedure that sets the slot number. This procedure must be called before any floating-point calculations are performed. The procedure is defined and called like this:

   ```
   procedure SetFPESlot (slot: integer); extern;

   ...

   begin
   SetFPESlot(3);
   ```

   Making this call will *not* hurt if you are using the standard SysFloat library, which contains a dummy version of this subroutine.

3. In addition, you can also use this directive:

   ```
   {$Float 1}
   ```

   This tells the compiler to generate code for the Innovative Systems FPE card. The other alternative, and the default, is

   ```
   {$Float 0}
   ```

which tells the compiler to generate code for SANE.

In ORCA/Pascal 2.0, the only thing this directive actually does is tell the compiler not to use certain shortcuts that generate direct calls to SANE for some simple binary math operations.

Incidentally, in ORCA/Pascal 2.0, once the compiler has been told to generate code for the 68881, no direct calls to SANE are made by any code generated by the compiler. All SANE dependencies lie in the libraries, which can be replaced for cross development to other machines.

ISO

The ISO directive is immediately followed by + or -. If + is used, an error is flagged whenever a feature is used which is not specifically required to be in the language by the ISO Pascal standard. The default is iso-. The iso directive is useful for detecting any ORCA/Pascal extensions that may be included in your source program, helping you to port the program to run under another compiler.

Examples:

```
{$ISO+}             { Use only ISO (Standard) Pascal}
{$ISO-}             { Allow ORCA/Pascal extensions}
```

Keep

The keep directive is followed by a string. The string contains the path name to use as the output file for the intermediate code produced by the compiler. The object module will be written to the keep file name. The path name can contain device numbers, prefix numbers, and can start with the directory walking characters **..** . If the path name contains only a file name, then the default prefix is assumed.

This directive must appear before the program token. Only one keep directive is allowed in a source file.

The keep directive is not normally used from the desktop environment.

From the text environment, if the keep directive is not used, you should use a keep parameter when compiling the program or set the shell keep variable to some default file name in order to cause the object module to be saved. If no keep file name is established by any of the three methods mentioned, no object module is created, and the link and execute steps cannot be performed. The keep parameter, keep variable, and the compilation process are discussed in Chapter 8.

Examples:

```
{$keep '../myprog'}
(*$keep '9:file3.exe'}
```

LibPrefix

When you use the `uses` statement in Pascal, the compiler looks for the file in a special subdirectory of the library prefix called ORCAPascalDefs; the actual prefix name used is 13:ORCAPascalDefs:.. This is where the interface files for the Apple IIGS tool kit, the ORCA shell, and GS/OS are located. You can also add your own interface files there. If the compiler does not find the interface file in ORCAPascalDefs, it will then look in the current prefix.

If you want the compiler to look somewhere else for interface files, you need a way to tell the compiler where to look. That is the purpose of the LibPrefix directive. Once used, the compiler will look in the directory specified by this directive for all future interface files. If you need to switch back to searching the standard ORCAPascalDefs directory, code this directive again with a null string for the prefix name.

You can specify a full or partial path name as the string.

If the unit you are trying to use has been placed in a library file in the system library prefix, or if it is in the current prefix, you do not have to do anything special when you link the program. If, however, you leave the unit as an object file, you will need to include it in the list of object modules when you link the program. See the section entitled Units for details.

List

The list directive is followed by + or -. If + is used, subsequent lines of the compiler listing are written to standard output. If - is used, subsequent lines are not included in the compiler listing. The default is -. You can also cause a listing to be produced from the shell's command line by including the +L option when compiling the source program. From the desktop, you can set a flag in the Compile dialog to create a listing. Unlike the list directive, these options does not give control over lines to be included in the listing – all lines in the program will be written to the listing file. You can use the list directive to select lines to be written to the listing file, as shown in the example below.

The list directive is generally used in conjunction with redirecting output to the printer to print only a portion of the program.

Examples:

```
{$list+}
{ These lines will be written to the listing file.}
program example;
var x, y: real;
(*$list- *)
(* This line won't be written to the listing file. *)
```

MemoryModel

The MemoryModel directive is used to tell the compiler whether you will be using the small memory model or the large memory model. The directive accepts an operand of 0 to indicate the small model, and 1 to request the large model. This directive must appear before the program token.

The CPU used on the Apple IIGS, the 65816, requires that code be segmented into pieces that can fit into one bank of memory, which is defined as 64K bytes in length. This is because the CPU cannot execute across a bank boundary. If the size of your combined program and data exceed 64K, you will start getting error messages from the linker during the link edit phase. The errors will report addresses outside of the current bank, or segments larger than 64K in length. You will know that you need the large memory model when these errors occur.

The small memory model assumes that all records, objects and arrays are smaller than 64K. This does not imply any restriction on the total amount of space used by your data; using dynamically allocated memory, you can easily allocate all of available memory using the small memory model, so long as no single array, record or object exceeds 64K.

The small memory model also assumes that all global arrays and variables are small enough to fit in a single 64K segment, and uses absolute addressing to access the global data.

Under the large memory model, the program is placed in its own 64K bank, and the data is divided into two other areas. All arrays and records are placed in a segment named ~ARRAYS, and any other global variables are placed in an area called ~GLOBALS. Since the data areas are not restricted to a single bank, the ~ARRAYS segment can be as large as available memory permits. The ~GLOBALS area is limited to a size of 64K bytes.

▲ **Warning** If any array or record exceeds 64K, even if the array or record is allocated dynamically with the new procedure or with NewHandle, you must use the large memory model. ▲

Should you find that you need more memory than 64K bytes for your program, you can use the segment directive, described later in this chapter, to further subdivide your program so that it will all fit into memory.

Tip In general, you should use the segment directive, not the large memory model, to split a program up into smaller pieces. The large memory model is only needed when you are using individual arrays, records or objects that exceed 64K, or when all of your global data exceeds 64K.

Examples:

```
{$MemoryModel 0}  {Use the small memory model – this is the default}
(*$MemoryModel 1*)       {Use the large memory model}
```

Names

The names directive is followed by + or -. If + is used, the compiler generates code to keep track of the name of the current procedure or function and the current line number. If a run-time error is encountered, the error message is followed by the line number and name of the procedure or function where the error occurred, as well as a trace back. The trace back shows the sequence of calls, given as procedure or function names, which resulted in the call to the subroutine where the error occurred. This list is given starting with the error and proceeds backward to the main program. The default is +.

You should be aware that keeping track of all of this information requires time and space. While in the development stages of writing your program, you will probably want to set trace back on. After the program is running, you will probably want to turn trace back off, so that the compiler may generate the most efficient code that it can.

Examples:

```
{$ names+}               { Set trace back on }
{$ names-}               { Set trace back off}
```

The following sample listing was produced with the RUN shell command, and with the list and names directives enabled in the program. The compiler listing is shown first, followed by the output from the linker. The program output then begins with the odd/even messages.

A run-time error was purposely introduced into the program. When i reaches 7 in the for loop of the main program, the call to the call procedure has an undefined case constant of 7. The trace back shows first the error that caused execution to stop. This is the "Jump to undefined case statement label" message. It next gives the line number in the program where the error occurred, as well as the name of the offending procedure. Finally, it lists the sequence of calls which resulted in the error. The trace back shows that die was called from call, and that call was called from the main program, named ~_PASMAIN in ORCA/Pascal.

```
ORCA/Pascal 1.2
Copyright 1987-1988, Byte Works, Inc.

   1 {$list+,names+,keep 'RunTimeBug'}
   2 program bug(output);
   3
   4 var
   5   i: integer;
   6
   7   procedure call(parm: integer);
   8
   9     procedure die(parm: integer);
  10
  11     begin
  12     case parm of
  13       1,3,5: writeln(parm,'   odd');
  14       2,4,6: writeln(parm,'  even');
  15       end; {case}
  16     end;
  17
  18   begin
  19   die(parm);
  20   end;
  21
  22 begin
  23 for i := 1 to 7 do
  24   call(i);
  25 end.

0 errors found


Link Editor 1.0a

Pass 1: .............................
Pass 2: .............................

There are 2 segments, for a combined length of $000009CF bytes.


       1   odd
       2  even
       3   odd
       4  even
       5   odd
       6  even
Jump to undefined case statement label
Error occurred at line 15 in procedure die

  Line  Name
  ----  ----
    19  call
    24  ~_PASMAIN
```

NBA

The NBA directive creates a program which has the correct calling sequence and environment for a HyperStudio New Button Action. This directive is discussed in detail in Chapter 4.

NewDeskAcc

The NewDeskAcc directive tells ORCA/Pascal that your program is a new desk accessory, and that the compiler needs to generate some special code. This directive must appear before the program token. The directive has seven parameters. The first four are the names of four procedures in your program that have special meaning in a desk accessory. The next two are the update period and event mask. The last is the name of your desk accessory, as it will appear in the Apple menu. The format is:

```
{$NewDeskAcc open close action init period eventMask menuLine}
```

open	This parameter is an identifier that specifies the name of the function that is called when someone selects your desk accessory from the Apple Menu. It must return a pointer to the window that it opens.
close	This parameter is an identifier that specifies the name of the procedure to call when the user wants to close your desk accessory. It must be possible to call this procedure even if open has not been called.
action	The action parameter is the name of a procedure that is called whenever the desk accessory must perform some action. It must declare two parameters. The first is a single integer parameter, which defines the action that the procedure should take. The second is an even record. See the Apple IIGS Toolbox Reference Manual for a list of the actions that will result in a call to this procedure.
init	The init parameter is the name of a procedure that is called at start up and shut down time. This gives your desk accessory a chance to do time consuming start up tasks or to shut down any tools it initialized. This procedure must define a single integer parameter. The parameter will be zero for a shut down call, and non-zero for a start up call.
period	This parameter tells the desk manager how often it should call your desk accessory for routine updates, such as changing the time on a clock desk accessory. A value of -1 tells the desk manager to call you only if there is a reason; 0 indicates that you should be called as often as possible; and any other value tells how many 60ths of a second to wait between calls.
eventMask	This value tells the desk manager what events to call you for. See the Apple IIGS Toolbox Reference Manual for details.
menuLine	The last parameter is a string. It tells the desk manager what the name of your desk accessory is. The name must be preceded by two spaces. After the name, you should always include the characters \H**.

Refer to Chapter 4 for more information about writing desk accessories.

Optimize

ORCA/Pascal is an optimizing compiler. Optimization is the process of improving the object code that would be generated by a simple compiler. The optimize directive accepts an integer operand; the value is used as a bit mask to turn individual optimizations on and off. In general, you should use either 0, which turns all optimizations off (and is the default) or -1, which turns all optimizations on. The bits used in ORCA/Pascal 2.0, and the optimizations they control, are:

bit	value	optimization
0	1	Intermediate code peephole optimization.
1	2	Native code peephole optimization.
2	4	Register use optimizations.
3	8	Common subexpression elimination.
4	16	Loop invariant removal and other loop optimizations.

This directive can appear anywhere in the program, but it is applied to an entire program level procedure or function and all of the procedures and functions imbedded in it at one time. For clarity, then, it is best to use the optimize directive between program level procedures or functions, or better still, to put a single optimize directive at the start of the source file.

Optimization takes time. If you want compilation to be faster, you can turn optimizations off. This is typically what you would do in a school environment, or during the development phase of a program.

When you want the compiler to produce the smallest, fastest program it can, you should turn optimizations on.

Examples:

```
{$ optimize 0}                  { Turn optimizations off }
{$ optimize -1}                 { Turn optimization on }
```

RangeCheck

The RangeCheck directive can be set to on (+) or off (-). It is used to force the compiler to check for the following:

1. Ensures subranges are valid.
2. Checks for integer math errors.
3. Checks if integers are equal to -maxint - 1.
4. Checks if long integers are equal to -maxint4 - 1.
5. Checks for real math errors.
6. Checks for use of nil pointers or objects.
7. Checks for array subscripts out of bounds.

8. Checks for stack overflows.
9. Ensures characters have ordinal values between 0 and 127, inclusive.
10. Ensures boolean values have ordinal values of 0 or 1.

This type of checking requires a great deal of extra time and space during compilation; hence, the default is for range checking to be off. You will typically want to enable range checking during the debugging phases of program development, and then disable the feature after the program is running.

The ISO standard requires that range checking be included in a Pascal compiler. In order to avoid using range checking, you will need to explicitly set `RangeCheck` off if you are also using the `{$ISO+}` directive.

Examples:

```
{$ iso+, rangecheck-} { Enforce ISO standard, turn range checking off }
{$ rangecheck+}       { Set range checking on }
```

RTL

The RTL directive tells the compiler to create a program that exits via the RTL assembly language instruction, rather than by making a call to the GS/OS procedure Quit. The directive must appear before the program statement to have any effect.

This directive is usually used to create Temporary Initialization Files (TIFs) and Permanent Initialization Files (PIFs). See the Apple IIGS Technical Notes for details about TIFs and PIFs.

Example:

```
{$ rtl }                 { Exit with an RTL }
```

Segment

The segment directive allows you to place procedures and functions in different load segments. A load segment is a block of executable code that is placed into memory by the loader. The directive accepts a string as an operand, which gives the name of the segment. The string can contain from one to ten characters, and the characters can be any printing characters. Note that segment names are case sensitive! That is, Seg1 is not the same segment name as seg1.

The segment directive is designed for use with programs that exceed 64K bytes in size, exclusive of the variables required by the program. All procedures and functions following a segment directive are placed into the named load segment. The same name can be reused in different parts of the program. The last segment named before the main program body causes the main program to be placed in that segment.

Examples:

```
program k;

procedure x;
begin
end;

{$ segment 'x'}

procedure z;
begin
{$ segment '33'}
end;

begin
end.
```

would cause procedure x to be placed in the 'blank' segment. This is the default name used by the loader if no segment directive has been used in the program, or if some parts of the program are not placed in a named segment with the segment directive. Procedure z would be placed in segment x, and the main program would be placed in segment 33.

StackSize

The StackSize directive is used to request that the compiler allot a specific size of stack. It accepts a single operand, the number of bytes that the stack may use, given as an integer. One of the first things that the compiler looks for is a StackSize directive, which must appear before the program token.

Local variables, parameters, and temporary variables allocated by the compiler are allocated from the stack. By default, the stack is 4K long. If you run out of stack space, you will get "Stack overflow" errors at run-time if range checking is on, or the program will crash if range checking is off.

Stack space is allocated from a special area of memory known as bank zero. The amount of memory actually available varies, depending on the version of GS/OS and tools in use, what program launcher was used, and so on. In general, you can get about 32K (32768 bytes) in any environment.

Stack space is actually allocated by creating a direct page segment. This causes the apparent size of your program to go up by the stack size, but no actual code has been added.

Examples:

```
{$ stacksize 4096}               { Default stack size }
{$ stacksize 10000}
```

Title

The title directive is followed by a quoted string. The string is the title that is to appear at the top of each page of the compiler listing. Title also causes page numbers to be printed at the top of each page of the listing.

Examples:

```
{$ title 'Project X:  TOP SECRET'}
```

ToolParms

This directive is used to force the compiler to create a function that is called with the protocols used by the Apple IIGS toolbox. It is needed whenever you write a function that will be called by a tool, such as when you create your own control using the Control Manager.

Normally parameters are placed on the stack, and the procedure or function is called. Upon return, function values are in the registers, or, in the case of real, double, comp or extended results, they are on the stack.

The tools use a different method. When a tool is called, space for the function result is placed on the stack, then the parameters are placed on the stack, and finally, the function is called. The value returned by the function is always on the stack. Tools that allow you to create functions that they will call expect your function to return the value on the stack. To cause ORCA/Pascal to do this, use the directive

```
{$ToolParms+}
```

before the function header. Be sure and place the directive

```
{$ToolParms-}
```

after the function, so that any functions in the rest of the program will use standard parameter passing methods.

Since Pascal expects any functions it calls to use the standard parameter passing convention, any function defined while this option is turned on cannot be called directly from Pascal. The function must be called only by the tools.

Example:

```
{$ToolParms+,DataBank+}

function MyFunc: integer;

begin
{function body goes here}
end;

{$ToolParms-,DataBank-}
```

XCMD

The XCMD directive creates a program which has the correct calling sequence and environment for a HyperCard IIGS XCMD. This directive is discussed in detail in Chapter 4.

Appendix A - ISO Conformance and Summary of Extensions

ISO Conformance

ORCA/Pascal V1.2 for the Apple IIGS complies with the requirements of level 0 of ISO 7185.

ORCA/Pascal V1.2 for the Apple IIGS compiles with the requirements of ANSI/IEEE 770 X3.97-1983.

Errors Not Caught

ISO 7185 defines an error in a somewhat peculiar way. An error is something which a perfect implementation of Pascal would catch, but that some compilers may not be able to catch because doing so would drastically increase the code size or running time of programs. A compiler can still be in full conformance to the standard, even if all of these errors are not caught, but all of the errors that are not caught must be listed. This section fulfills that requirement.

1. It is an error to access any component of a variant record if that part of the record is not active.
2. It is an error to allocate space for a variant record using the form `new(p,V1,V2,...)` and then to use the variable in an assignment or as an actual parameter to a procedure or function call.
3. It is an error to change the value of a file variable `f` when a reference to its buffer variable `f^` exists.
4. It is an error to try to reference a variable through an undefined pointer. If the pointer has been deallocated using dispose, this error will be caught. It is not always caught if the pointer has never been allocated. The error is never caught if two pointers point to the same area, and one is disposed of, and then the other is accessed.
5. It is an error to try to dispose of a heap area while a reference to it exists.
6. When a variable has been allocated with the form new(p,V1,V2,...), where V1, V2, …, indicate variant portions of a record, it is an error to try to change one of the variant parts that were specified (although a variant part at a deeper level can be changed).

   For example, if a variant record had been defined as

```
record
   case b: boolean of
      true:  (i: integer);
       false: (r: real)
    end;
```

   then an allocation of memory for the record using the statement

```
new (p, true);
```

would result in the variable i being defined and a total of four bytes of memory being allocated. A later assignment such as

```
p^.b := false;
```

is clearly an error. This type of error will not be caught by ORCA/Pascal.

7. It is an error to allocate a heap variable with one parameter list like new(p,V1,V2), and then deallocate with another, like dispose(p,VA,VB). The parameter lists must be identical.
8. It is an error for the result of pred or succ to not exist. ORCA/Pascal will only catch this error if the result is assigned to a variable of the same type as that upon which the pred or succ operation is performed. For example, consider the following code fragment:

```
digits = 0..9;
d, e: digits;
i: integer;

d := 9;
i := succ(d);            {would not be caught – d and i are of }
                         {different types}

e := succ(d);            {would be caught – e and d are of }
                         {the same type}
```

9. It is an error for the result of a function call to be undefined. ORCA/Pascal will only detect this error if there are no assignments in the function body. If at least one assignment exists, but there is a path through the function that does not execute the assignment, the error will not be caught.

Implementation Defined

The ISO standard specifies some items which can change from one implementation of Pascal to another, but for which some definition must exist. In ORCA/Pascal, these are:

1. Maxint = 32767.
2. Char variables can be assigned ordinal values in [0..127]. The characters associated with these ordinal values are defined by the ASCII character set.
3. Real numbers can range from -3.4e38 to -1.2e-38 and from 1.2e-38 to 3.4e38. Computations are accurate to seven decimal digits. Double-precision real numbers can range from -2.3e-308 to -1.7e+308 and from 2.3e-308 to 1.7e+308. Computations are accurate to fifteen decimal digits. Extended-precision real numbers can range from -1.7e-4932 to -1.1e+4932 and from 1.7e-4932 to 1.1e+4932. Computations are accurate to

nineteen decimal digits. Comp numbers can range from -9.2e18 to 9.2e18; they are always integer values. Computations are accurate to eighteen decimal digits.

4. Sets may contain up to 2048 elements, with ordinal values in [0..2047].
5. Default field widths for the write and writeln procedures are:

byte	8
integer	8
longint	16
boolean	8
real	16
double	16
extended	16
comp	16

6. Exponents are written as the character `e` followed by a three-digit signed integer value, as in

```
3.141593e+000
```

7. The page procedure writes the ASCII form-feed character to the specified file. The ordinal value of the form-feed character is 12.
8. Variables listed in the program header are not bound to external files or values, except for the required identifiers input, which is bound to standard input (usually the keyboard) and output, which is bound to standard output (usually the CRT screen), and the identifier erroroutput, which behaves like output but sends characters to the error output hook.
9. Use of the Standard Pascal procedures reset or rewrite on input, output, and erroroutput is ignored.

Extensions to the ISO Pascal Standard

Compiler Directives

Compiler directives give you control over the way the compiler processes your program. Compiler directives appear in the source as if they were comments, except that the character after the comment character is a $ character. No spaces can separate the opening comment character from the $ character. See Chapter 23 for descriptions of the directives, as well as details on the syntax and error checking used while processing compiler directives, including how to code more than one in a single comment.

```
{$Append 'filename'}
{$ClassicDesk 'name', start, shutdown}
{$CDev name}
{$Copy 'filename'}
```

```
{$DataBank+}                        {$DataBank-}
{$Dynamic 'segmentName'}
{$Eject}
{$ISO+}                             {$ISO-}
{$Keep 'filename'}
{$LibPrefix 'directory'}
{$List+}                            {$List-}
{$MemoryModel 0}                    {MemoryModel 1}
{$Names+}                           {$Names-}
{$NewDeskAcc open close action init period eventMask menuLine}
{Optimize -1}                       {Optimize 0}
{$RangeCheck+}                      {$RangeCheck-}
{$RTL}
{$Segment 'segmentName'}
{$StackSize n}
{$ToolParms+}                       {ToolParms-}
{$Title 'any string'}
```

Additional Language Features

The case statement has been extended to allow the use of an otherwise clause. The otherwise clause is executed if the input value for the case statement is not specified as a case label. For example, the loop

```
for i := 1 to 5 do
  case i of
    1: writeln('1st');
    2: writeln('2nd');
    3: writeln('3rd');
    otherwise: writeln(i:1,'th');
    end; {case}
```

would write

```
1st
2nd
3rd
4th
5th
```

to the screen. The otherwise clause can occur anywhere in the body of the case statement, but only one otherwise is allowed. Under ORCA/Pascal otherwise is a reserved word, unless the {$ISO+} directive has been used.

ORCA/Pascal supports a uses statement to import declarations from a separate file. Note that under ORCA/Pascal uses is a reserved word, unless the {$ISO+} directive has been used.

The extern, ProDOS, Tool, UserTool and Vector directives can be used to control importing and exporting of global variables.

Additional Types

Four additional predefined types exist in ORCA/Pascal. Longint is an extended form of the type integer. Longint variables require four bytes of storage. They can represent integer numbers in the range -2147483647 to 2147483647. The companion predefined constant maxint4 is also defined, with a value of 2147483647. With a few minor exceptions, longint constants and variables can be used anywhere that integer constants and variables can appear. For the most part, you can think of longint as if it was the Standard Pascal type integer, and integer as if it is a predefined subrange of longint.

Byte is a subrange of the standard type integer. A value of type byte can range from 0 to 255, and occupies one byte of storage.

There are three predefined real types in addition to the required type real; they are double, extended and comp. Variables and constants of type double, extended and comp can be used anywhere that a real variable or constant appears.

Finally, ORCA/Pascal supports strings. Strings are actually not a new type, but rather a mnemonic for packed array [n..m] of char, with n 0 or 1, and m greater than one. If no length is specified, a maximum length of eighty characters is assumed. Any length in the range 2..maxint is supported.

ORCA/Pascal supports two internal formats for strings. Either format may be used interchangeably within Pascal; it is only when you are dealing with the toolbox, or for some other reason need to use one format or the other, that you must be concerned with which format is used. Standard Pascal strings are implemented in ORCA/Pascal along the lines of strings in the C programming language. That is, an ASCII zero (null character) can be placed anywhere within a string to mark its end. The length of a string is the number of characters up to but not including the null character, or it is the maximum size given when the string was defined, whichever is shorter. Strings with a leading length byte are also supported. This format limits the length of strings to 255 characters, while Standard Pascal strings can be up to 32767 characters long.

See the section on strings, below, for a package of extensions that give a great deal of added power to strings in ORCA/Pascal.

Additional Operators

Six operators have been added to allow bit manipulations of integers, bytes, and long integers. They are:

&	Bitwise and.
|	Bitwise or.
!	Bitwise exclusive or.
~	Bitwise not.
<<	Shift left.
>>	Shift right.

The operators work exactly like their C language counterparts.

ORCA/Pascal also supports the exponentiation operator **. This returns an extended result.

Finally, the @ operator can be used to return the address of a variable, function, or procedure, or the address of an element of an array or record. The type of the result is assignment compatible with any pointer type.

The precedence of these operators, compared with the operators from Standard Pascal, are:

not	~	**	@				
div	mod	and	*	/	&	<<	>>
or	+	-	|	!			
in	=	<>	<	>	<=	>=	

Error Output

In addition to the predefined files input and output, ORCA/Pascal also supports the predefined file erroroutput. Its characteristics are identical to the characteristics of output. The difference is that characters written to output are sent to the Apple IIGS standard output hooks, while characters written to erroroutput are written to the Apple IIGS error output hooks. You can write error messages to erroroutput, and they will appear on the console even if output has been redirected. You can also separately redirect erroroutput.

Strings

ORCA/Pascal has extended the definition of strings to allow reading a string using the read and readln procedures, assigning strings of different lengths, support for assigning a zero length string, and changing the current length of a string.

There are two internal formats for strings, either of which may be used anyplace Pascal allows a string. One format is a length byte followed by up to 255 characters. This format is coded as

```
packed array [0..n] of char
```

where n is in the range 2..255. The other format is the format used in ISO, ANSI, and Jensen & Wirth – in other words, Standard Pascal strings. Standard strings are coded as

```
packed array [1..n] of char
```

where n is in the range 2..maxint. Some extensions to Standard Pascal strings are also supported. The specific addition made to Standard Pascal strings are:

1. A string can be assigned to a string that was defined with a different length. If the target string is shorter than the source string, extra characters are truncated. If the target string is longer, a chr(0) character is inserted after the last character copied from the shorter string.

2. The character chr(0) is interpreted to mean the end of the string. The three places where this has an effect are:

   a. When a string is written to an output device, encountering a chr(0) has the effect of canceling the output of the remainder of the string.
   b. When the length of a string is computed using the length function, scanning stops when a chr(0) is encountered.
   c. A chr(0) character is inserted after the last character read from a file using read or readln, and after the last character copied when a short string is copied into a longer one.

3. The null string '' can be assigned to a string. The affect is to set the first character of the string to chr(0).
4. A character can be assigned to a string. In that case, the character is placed in the first element of the string, and chr(0) becomes the second element of the string.
5. A string can be read from a file using the procedures read and readln.

The two string formats may be used interchangeably. Strings of one format may be assigned to strings of another format, and the two formats may be mixed freely when using the built-in procedures and functions that handle strings. When passing parameters, normal assignment compatibility does have an effect, however. Strings passed as parameters must be of the same format as the parameter; i.e., they must have the same length and be of the same type.

Predefined Procedures and Functions

This section gives an overview of the procedures and functions available from ORCA/Pascal that are not included with Standard Pascal, or that have additional features not provided in the Standard Pascal implementation.

For more details on any of these functions, or on the functions and procedures of Standard Pascal, see Chapter 22, or refer to the index or table of contents for specific page numbers.

```
function arccos (x: real): real;
function arcsin (x: real): real;
function arctan2 (x, y: real): real;
procedure close (f: file);
procedure CommandLine (var string[size]);
function cnvds (x: extended; width, digits: integer):
                string[size];
function cnvis (x: integer): string[size];
function cnvrs (x: extended; width, digits: integer)
                string[size];
function cnvsd (str: string[size]): extended;
function cnvsi (str: string[size]): integer;
function cnvsl (str: string[size]): longint;
function cnvsr (str: string[size]): real;
function concat (str1: string[size1]; str2: string[size2]):
                 string[rsize];
function copy (str: string[size]; index, count: integer):
               string[rsize];
procedure delete (str: string[size]; index, count: integer);
procedure EndDesk;
procedure EndGraph;
procedure halt (errorCode: integer);
procedure insert (str1: string[size1]; str2: string[size2];
                  index: integer);
function length (str: string[size]): integer;
function member (obj: objectType; otype: objectType): boolean;
procedure open (f: file; name: string[size]);
function ord4 (x: <any ordinal value>): longint;
function pointer (x: <any pointer>):<anonymous pointer>;
function pos (target: string[tsize]; source: string[size]):
              integer;
function random: real;
function randomdouble: extended;
function randominteger: integer;
function randomlongint: longint;
procedure reset (f: file; name: string[size]);
procedure rewrite (f: file; name: string[size]);
function round4 (x: real): longint;
procedure seed (x: integer);
procedure seek (f: file; loc: integer);
procedure shellid (var string[size]);
function SizeOf (value): integer;
procedure StartDesk (resolution: integer);
procedure StartGraph (resolution: integer );
procedure SystemError (errno: integer);
function tan (x: real): real;
function ToolError: integer;
function trunc4 (x: real): longint;
function userID: integer;
```

Appendix B - Error Messages

The errors flagged during the development of a program are of three basic types: compilation errors, linking errors, and execution errors. Compilation errors are those that are flagged by the compiler when it is compiling your program. These are generally caused by mistakes in typing or simple omissions in the source code. Compilation errors are divided into four categories: those that are marked with a caret (^) on the line in which they occurred; those where the exact position of the error cannot be determined or is not well defined, such as declaring a label but then not using it in the body of a block; those which are due to the restrictions imposed by the ORCA/Pascal compiler; and those which are so serious that compilation cannot continue. Each of the error types is treated in its own subsection of compilation errors.

Linking errors are those that are reported by the linker when it is processing the object modules produced by the compiler. These are typically caused by lack of memory for the object code or data, or by incorrectly linking files when separate compilation has been used. If you receive "Out of memory" messages from the linker, try using the large memory model available with the compiler, or you can break up your program into different load segments. When the linker issues "Unresolved reference" or "Duplicate reference" errors, you have probably made a mistake in your external declarations.

Execution errors occur when your program is running. These can be detectable mistakes, such as division by zero, or can be severe enough to cause the computer to crash, such as accessing memory in unexpected ways, as with pointer variables containing invalid addresses.

Error levels are associated with compilation and linking. Compiler error levels are explained at the beginning of the section describing the errors you can receive during compilation; linker error levels are described at the beginning of the section describing linker errors.

Compilation Errors

')' Expected

The compiler expected to find a right parenthesis but did not see one. Compilation continued as though a ')' had been inserted into the source file.

':' Expected

The compiler expected to find a colon but did not see one. Compilation continued as though a ':' had been inserted into the source file.

'(' Expected

The compiler expected to find a left parenthesis but did not see one. Compilation continued as though a '(' had been inserted into the source file.

'[' Expected

The compiler expected to find a left bracket but did not see one. Compilation continued as though a '[' had been inserted into the source file.

']' Expected

The compiler expected to find a right bracket but did not see one. Compilation continued as though a ']' had been inserted into the source file.

';' Expected

The compiler expected to find a semicolon but did not see one. Compilation continued as though a ';' had been inserted into the source file.

'=' Expected

The compiler expected to find an equal sign but did not see one. Compilation continued as though a '=' had been inserted into the source file.

',' Expected

The compiler expected to find a comma but did not see one. Compilation continued as though a ',' had been inserted into the source file.

'.' Expected

The compiler expected to find a period but did not see one. Compilation continued as though a '.' had been inserted into the source file.

':=' Expected

The compiler expected to find an assignment symbol but did not see one. Compilation continued as though a ':=' had been inserted into the source file.

'..' Expected

The compiler expected to find a range symbol but did not see one. Compilation continued as though a '..' had been inserted into the source file.

Actual Parameter Must Be A Variable

A parameter to a procedure or function has been declared as a var parameter, but an attempt has been made to pass a constant or expression. Values passed as var parameters must be variables, array elements, or record elements.

Again Forward Declared

A procedure or function has been declared as forward more than one time, or has been declared as forward after being declared in the interface part of a unit.

Assignment Of Files Not Allowed

A file variable has been assigned to another variable. This is not allowed in Pascal.

Assignment To Formal Function Is Not Allowed

A function name was passed as a parameter to another function or procedure, and an attempt was then made to assign a value to the passed parameter..

Assignment To Function Identifier Not Allowed Here

An attempt has been made to assign a value to a function, but the assignment statement did not occur within the body of the function.

Assignment To Standard Function Is Not Allowed

An attempt was made to assign a value to one of the built-in functions. These can be either the standard ISO functions, or one provided with ORCA/Pascal. For example, sin := x.

Base Type Must Be Scalar Or Subrange

When defining a set, an element of the set has been specified which is of a type that is invalid for membership, such as a pointer or another set.

Base Type Must Not Be Real

When defining a set, the base type has been given as real, double, comp or extended. Real numbers cannot be used as elements of sets in Pascal.

'Begin' Expected

The compiler expected to find the keyword `begin` but did not see it. Compilation continued as though begin had been inserted into the source file.

Body Must Appear in Implementation Part

A function or procedure in the interface part has a body (the begin and end, plus the statements in between). The body for the procedure or function must be moved to the implementation part of the unit.

Cannot Modify Control Variable

An attempt was made to assign a value to the looping control variable in a `for` statement, as in

```
for k := 1 to 5 do
   k := 10;
```

Cannot Use As Formal Parameter

A parameter has been declared as a var parameter to a function or a procedure, and an attempt has been made to pass an illegal variable as the parameter. The kinds of variables which cannot be passed as var parameters include elements of packed arrays, tag fields of variant records, and the control variables of for loops, from within the for loop itself. Each of the parameters passed below to the procedure `WrongCall` are illegal.

```
type clothes = record {gender is tag field}
  case gender: integer of
    female: (blouse, skirt: integer);
    male: (shirt, pants, jacket: integer);
  end;

var
  letters: packed array [1..20] of char;
  k: integer;

begin
for k := 1 to 5 do
  {k is control variable of for loop}
  WrongCall (k, letters[3], gender);
```

where WrongCall has been defined as:

```
procedure WrongCall(var x: integer; var ch: char; var y: integer);
```

Casted Expression Must Be Scalar or Pointer

A value that cannot be safely type cast has been type cast, or a type cast to a type that cannot be used has been attempted.

Code Generation Error

This error should only occur in the presence of other errors. Correcting the other errors will cause this one to go away, too. If this error occurs by itself, report it as a bug to the Byte Works.

Compiler Error

This error generally indicates that the symbol table has been corrupted, due to the compiler becoming confused after an earlier error. You should not receive this error except after receiving at least one other error. Correcting the other error(s) will make this one go away, too. Should the error occur by itself, report it as a bug to the Byte Works.

Note that this error is sometimes reported first, and an error the compiler detects a few tokens later is the cause.

Control Var Must Be Declared At This Level

An attempt has been made to use a looping variable in a for loop which was not declared in the procedure or function that contains the for loop.

Digit Expected

A number has been found which is immediately followed by a character, with no intervening spaces. Pascal requires that identifiers, reserved words, and numeric constants be separated from one another by at least one space.

'Do' Expected

The compiler expected to find the keyword do but did not see it. Compilation continued as though do had been inserted into the source file.

Duplicate Label

This error should only occur in the presence of other errors. Correcting the other errors will cause this one to go away, too. If the error occurs by itself, report it as a bug to the Byte Works.

Element Expression Out Of Range

An element in a set constant is outside of the range 0..2047.

'End' Expected

The compiler expected to find the keyword end but did not see it. Compilation continued as though end had been inserted into the source file.

Error In Base Set

A set was defined in such a manner that the base type was out of bounds for its ordinal range. For example, set of [2000..3000] is invalid because the ordinal value of each set element must be within the range 0..2047 under ORCA/Pascal.

Error In Constant

An error was detected in the construction of a constant. This is typically caused by a typographical error when coding a constant value.

Error In Declaration Part

The declaration section of a block has been coded in such a way that the compiler thought that the declaration section was complete, yet it found that the section continued. Check that you have listed your declarations in the correct order (uses, label, const, type, var, subroutine heading, begin).

Error In Factor

An error was detected in the construction of a factor. This is typically caused by a typographical error when coding an unsigned constant value, an identifier, a variable, a function call, an expression enclosed in parentheses, or a subrange specification.

Error In Field-List

A record specification was incorrectly constructed.

Error In Parameter List

A parameter list for a procedure or function call was incorrectly constructed. This is usually caused by omitting a closing parenthesis.

Error In Simple Type

An incorrect array or set specification was given. After encountering the key words `array` or `set of`, followed by a left square bracket, the compiler expected to see a type identifier, a list of identifiers enclosed in parentheses, or a subrange. Check the specification enclosed in square brackets.

Error In Type

A syntax error was detected during compilation of the type declaration section of the block. Check the type declaration.

Error In Type Of Standard Function Parameter

A parameter of the wrong type was used when calling a built-in function. For example, `sin(setValue)` rather than `sin(realValue)`. The built-in routines include those prescribed by Standard Pascal as well as the extended functions provided with ORCA/Pascal.

Error In Type Of Standard Procedure Parameter

A parameter of the wrong type was used when calling a built-in procedure. For example, `reset(stringType)` rather than `reset(filePointer)`. The built-in routines include those prescribed by Standard Pascal as well as the extended procedures provided with ORCA/Pascal.

Error In Uses

A syntax error was detected during compilation of a `uses` file. Check file referred to in the `uses` directive.

Error In Variable

An error was detected in the construction of a variable. This is typically caused by a typographical error when coding a variable declaration.

Expression Is Not Of Set Type

The right-hand operand of an `in` operator was not a set.

Extern Allowed At Program Level Only

A procedure or function declaration, with an attached `extern` directive, was found, and the procedure or function is not callable from the main program. External subroutines can only be declared in the declaration section of the main program.

External Variable Cannot Be Procedure Or Function

In the program declaration section of the source file, an identifier was used which was later declared as the name of a procedure or function. For example,

```
program ex (wrong);
procedure wrong;
```

F-Format For Real Only

The output format for a variable or expression in a write statement has been specified using the `:integer :integer` notation, but the variable or expression is not of type real. For example, `write(4:3:2)` would generate this error, since the value to be written (4) is an integer, and Pascal forbids the specification of the number of digits to be written for an integer value. `Write(4.0:3:2)` is legal, however, because the value to be written is a real number.

File Cannot Contain Another File

A file variable has been declared with a type that contains another file variable. Files cannot contain other files.

File Comparison Not Allowed

A comparison of two files was encountered. This type of comparison is not allowed in Standard Pascal.

File Value Parameter Not Allowed

A file pointer variable was passed to a procedure or function as a value parameter. Only file pointers which are var parameters may be passed to procedures or functions.

Forward Declared Function; Cannot Repeat Type

A function has been declared with a forward directive attached to the declaration. The same function has been encountered later in the block with the return type repeated. The second appearance of the function should not include the return type.

Forward Declared; Repetition of Parameter List Not Allowed

A procedure or function has been declared with a forward directive attached to the declaration. The same procedure or function has been encountered later in the block with the parameter list repeated. The second appearance of the procedure or function definition should not include the parameter list.

Forward Reference Not Resolved

A procedure or function was declared with an attached forward directive, but the body of the procedure or function was never found.

Function Result Type Must be Scalar, Subrange, or Pointer

The type to be returned by a function was of an inappropriate type, such as a set or file. For example, you cannot declare a function as

```
function f: set of 1..10;
```

in Standard Pascal.

Further Errors Suppressed

ORCA/Pascal reports a maximum of ten errors per line of source code. All errors will be counted, however, and reflected in the total shown at the end of the compilation. If more than ten errors are found in a single source line, this is the last error message shown for the line.

Identifier Declared Twice

An identifier has been used twice in the declaration section of a block. Check the names you have given to types, variables, and constants in this block.

Identifier Expected

The compiler expected to find an identifier, but found some other token. This is generally caused by a typographical error.

Identifier Is Not Of Appropriate Class

An identifier has been incorrectly given as the name of a type. This can be caused by a typographical error when coding the identifier, or omitting the identifier in the type declaration section of a block.

Identifier Not Declared

An identifier has been used in a statement without declaring the identifier in the declaration section of a block. This can be caused by a typographical error when coding the identifier, failure to define a variable, or omission of a function or procedure declaration.

If this error occurs in a uses statement, the compiler is telling you that the uses statement uses a type that has not been defined. This could occur, for example, if the segment you are including included another segment and used some definitions from that segment, then you use the segment without first using the other segments it needs.

Illegal Goto

A goto statement was found which attempted to jump to a label defined within a structure. Standard Pascal forbids branching to a label which is attached to a line contained within a statement of any kind. For example, you cannot go to a label defined within a for loop, a begin/end block, or a case statement.

Illegal Parameter Substitution

A type conflict exists between the parameter(s) in a procedure or function call, and the type specified when the procedure or function was defined.

Illegal Symbol

The compiler found a character that did not belong in the place it was found. This can be caused by an invalid character anywhere in the source file, such as %, or a symbol used in the wrong context, such as an underscore by itself rather than as part of an identifier. The error can also be issued if a string is improperly formed, or if you have coded an incorrect symbol as the operand of a directive, such as a -1 for a directive which expected either a + or a -.

Illegal Type Of Expression

An expression has been used in a case or for statement which is not an ordinal type. For example, `case i of 3.4` or `for k := 12.9`. Ordinal types include char, boolean, integer, enumerations, and subranges.

Illegal Type Of Loop Control Variable

The loop variable in a for loop was of the wrong type. The loop variable must be an ordinal type: boolean, integer, char, enumeration, or subrange.

Illegal Type Of Operand(s)

An operation was encountered whose operand(s) were of the wrong type, such as `real div integer`.

'Implementation' Expected

The compiler expected to find the reserved word implementation, signaling the start of the implementation part of a unit, but encountered some other symbol.

Implementation Restriction

ORCA/Pascal imposes the following restrictions on sizes of variables:

1. Variant records are limited to 32767 tags.
2. Arrays cannot be larger than 64K bytes long.
3. Records cannot be larger than 64K bytes long.

This error occurs when one of these limits is exceeded.

Implementation Restriction: String Space Exhausted

Each program level procedure or function is limited to 8K of total string space. String space is used by string constants and by debug code.

Incompatible Subrange Types

The type of the first constant does not match the type of the second constant in a subrange declaration, such as `true..4`.

Incompatible With Tagfield Type

The type of an expression in a case statement of a variant record definition does not match the type of the variable used as the selector. For example: `case a: boolean of 4`. The boolean value `a` cannot be equal to `4` – it can only be either `true` or `false`.

Index Type Is Not Compatible With Declaration

A subscript was applied to an array, and the subscript is not of the type defined for the array.

Index Type Must Be Scalar Or Subrange

An array subscript has been used which is of the wrong type. Index types for arrays must be ordinal types (integer, boolean, character, enumeration, or subrange) and cannot be of type longint.

Integer Constant Exceeds Range

A constant integer was found which is outside of the range -maxint4..maxint4 (-2147483647 to 2147483647).

Integer Expected

The compiler expected to find an integer but encountered some other token. This is generally caused by a typographical error when coding a subrange.

'Interface' Expected

The compiler expected to find the reserved word interface, signaling the start of the interface part of a unit, but encountered some other symbol.

Label Space Exhausted

ORCA/Pascal places all of the labels (procedure and function names, user-defined type, variables, etc.) defined in any particular program-level procedure or function into one area known as the label table. Generally, there is enough room in the label table for a single procedure or function that is approximately 2000 lines long. When this error is received, you should break up the procedures and functions, or the main program itself if necessary, into smaller procedures and functions that can be called from the main program.

Label Type Incompatible With Selecting Expression

A case constant is of a different type than that used in the defining expression, as in `case k of 'a':` where `k` has been defined to be of type integer.

Low Bound Exceeds High Bound

The first constant is larger than the second constant in a subrange declaration.

Methods Must Be Declared at the Program Level

Methods cannot be imbedded in other procedures or functions. They can be declared at the program level of a program, or in the interface or implementation part of a unit.

Misplaced Directive

A directive was used in a place or in a situation in which it was not valid. See the description of the directive for details about how it can be used.

Missing Corresponding Variant Declaration

A call to the standard procedures `new` or `dispose` has included a reference to an undefined variant field. For example, `new (p, V1, V2)`, where `V2` is not a part of the record pointed to by the variable `p`.

Missing 'Input' In Program Heading

A read, readln, eof, or eoln call has been issued which does not use a file variable as its first parameter, and the special file `input` was not defined in the program heading of the source file. If you intend for input to be entered from standard input, you must include the keyword `input` in the program heading. If `input` does not appear in the program heading, you must code a file variable as the first parameter of a read, readln, eof or eoln call.

Missing 'Output' In Program Heading

A write, writeln, or page call has been issued which does not use a file variable as its first parameter, and the special file output was not defined in the program heading of the source file. If you intend for output to be sent to standard output, you must include the keyword output in the program heading. If output does not appear in the program heading, you must code a file variable as the first parameter of a write, writeln, or page call.

Missing Result Type In Function Declaration

A subroutine has been declared as a function, but the type of value returned by the function has been omitted from the declaration.

Multideclared Label

The same label has been defined more than once in a label declaration statement.

Multidefined Case Label

The same case label has been used more than once in the body of a case statement.

Multidefined Label

The same label has been used more than once in the body of a procedure or function.

Multidefined Record Variant

The same variant field name has been used more than once within the case statement of a variant record declaration.

No Assignment To Function Found

An assignment of a value to the function identifier was omitted from the body of the function.

No Such Field In This Record

A variable name was followed by a period, but the identifier after the period was not declared as being an element of the record named by the variable preceding the period.

Not A Known Object

An attempt has been made to define a method for an object class, but the object class has not been declared.

Not ISO Standard

The iso+ directive has been coded, and an attempt to use an ORCA/Pascal extension has been made.

Number Expected

The compiler expected to find a number but encountered some other token. This is generally caused by a typographical error when coding a real or integer constant.

Number Of Parameters Does Not Agree With Declaration

A procedure or function has been called with a parameter list that is longer or shorter than the parameter list given when the procedure or function was declared.

Objects Cannot Have a Variant Part

Unlike records, objects and classes cannot have a variant part, but a case statement which appears to start a variant part has been found in an object class declaration.

Objects Must Be Declared as a Named Type

Object classes must be declared in a type statement at the program level, but an object class declaration was found on another context.

Object Expected

An object class declaration contains an opening parenthesis that appears to be the start of an inherited object sequence, but the next symbol is not an identifier.

'Of' Expected

The compiler expected to find the keyword `of` but did not see it. Compilation continued as though `of` had been inserted into the source file.

Only Extern, Forward, ProDOS, Or Tool Allowed In Uses File

A complete procedure or function was encountered within a uses file. These types of files require that references to procedures or functions be declarations only; the body of a procedure or function cannot be included in a uses file. The ORCA/Pascal directives which indicate that a subroutine is external to the file in which they are declared include extern, ProDOS, Tool, UserTool and Vector. The directive denoting that a subroutine body will be given later in the source file is forward. The procedure or function headings included in a uses file must be declared as either extern, forward, Tool, UserTool, Vector or ProDOS.

Only Tests Of Equality Allowed

A comparison of two pointers was detected, and the comparison operator was neither = nor <>.

Parameter Size Must Be Constant

The standard procedure new or dispose has been called with a parameter that should have been the name of a variant field, but instead has been coded as an expression, as in new (p,4).

Pointer Cannot Be Resolved

A variable has been declared as a pointer to an identifier, but the identifier was not declared later in the same block. For example, `type fptr = ^foo` requires that an identifier called foo be declared later in the block.

Pointer References A File

A pointer variable has been assigned the address of a file variable. For example, `fptr := ^foo;` where foo is a file variable.

Previous Declaration Was Not Forward

A procedure or function has been declared twice, and the first declaration did not include a forward directive.

'Program' Expected

The compiler expected to find the keyword program but did not see it. Compilation continued as though program had been inserted into the source file.

Quoted File Name Expected

A copy directive, append directive, or a uses statement was given which did not include a file name.

Result Type of Function Does Not Agree with Declaration

The value assigned to a function is not of the same type as that specified when the function was defined.

Set Element Type Must Be Scalar Or Subrange

A set constant has been defined which contains an element of the wrong type, such as real.

Set Element Types Not Compatible

A set constant has been defined which contains elements of different types, such as `[4,'a']`.

Sign Not Allowed

A plus or minus sign preceded a constant value which was not a real or integer value, or two sign symbols in a row were detected.

Strict Inclusion Not Allowed

A comparison of two sets was detected and the comparison operator was either < or >. Set `s1` < set `s2` implies that `s1` is completely included in `s2`; this type of comparison is not allowed in Standard Pascal.

String Constant Must Not Exceed Source Line

A string constant has been found which continues for more than one line of source code. This is usually caused be forgetting to close the string with an ending quote mark.

String Expected

A compiler directive requires a string parameter, but either found some other symbol, or did not find any other symbols in the directive at all.

String Space Exhausted

Too many strings have been defined in any one procedure or function. ORCA/Pascal limits the total amount of characters occupied by all strings in one procedure or function to 8000.

Subrange Bounds Must Be Scalar

A subrange was found whose defining bounds were not of a scalar type. In Pascal, a scalar type is synonymous with an ordinal type: char, boolean, integer, enumeration or subrange.

Subrange Exceeded

An attempt has been made to assign a value outside of the defined bounds for a subrange variable. For example,

```
    i: [1..10];
    . . .
    i := 11;
```

This error will only occur if the `RangeCheck` directive has been enabled.

Tagfield Type Must Be Scalar Or Subrange

The type of the variable used as the selector for a variant record is real, double, comp or extended. Only selectors of type char, boolean, or integer may be used.

'Then' Expected

The compiler expected to find the keyword `then` but did not see it. Compilation continued as though `then` had been inserted into the source file.

There Is No Method to Inherit

The inherited reserved word has been used in a method, but the identifier that followed was not the name of a method that was available to be inherited.

'To' Expected

The compiler expected to find the keyword `to` but did not see it. Compilation continued as though `to` had been inserted into the source file.

Too Many Nested Procedures And/Or Functions

ORCA/Pascal allows nesting of procedures and functions to a level of ten deep, counted from the main program level.

Too Many Nested Scopes Of Identifiers

A with statement, defined at the program level, can have a maximum of 20 record identifiers. One is subtracted from this maximum for each level of procedure or function nesting. That is, if a with statement is used in a procedure that is nested five levels deep from the main program level, then the with can have a maximum of 15 record identifiers.

Type Conflict

An expression has been given in a case or for statement that is not of the same type as the case constant or for looping variable. For example, `case k of 3`, where `k` is of type boolean, or `for m := 1 to 5`, where `m` is of type char.

Type Conflict Of Operands

A binary operation is to be performed but the operands for the operation are of incompatible types, such as `string <= 4`.

Type Must Not Be Real

A real constant was used in a declaration where real values are not permitted, such as in the range specification for a set.

Type Of Operand Must Be Boolean

A not operator was found whose operand was not of type boolean.

Type Of Variable Is Not Array

A subscript was applied to a variable which has not been defined as an array.

Type Of Variable Is Not Record

A variable name was encountered which was followed by a period, but the variable was not defined as being of type record.

Type Of Variable Must Be File Or Pointer

Assignment to a variable immediately followed by a pointer symbol, as in `f^ :=`, was detected, but the variable was not defined as being of type file or pointer.

Type Of Variable Must Be Object

An object class declaration contains an identifier in parenthesis, but the identifier is not an object, or it has never been declared.

Undeclared External Variable

In the program declaration section of the source file, an identifier was used which was never declared as a variable. For example, `program ex (wrong);` where wrong was not declared as a variable in the main program.

Undeclared Label

A label has been referenced in a goto statement, but the label was not defined in a label declaration statement, or a label has been coded in the body of a procedure or function but was not defined in a label declaration statement.

Undeclared Method

An attempt has been made to override or define a method, but the method was not declared in the object class.

Undefined Label

A label was referenced in the declaration section of a block, but was never used in the statement portion of the block.

Unexpected End Of File

The end of the source file was encountered before the end of the program was detected. This usually indicates invalid construction of the main program.

'Until' Expected

The compiler expected to find the keyword `until` but did not see it. Compilation continued as though `until` had been inserted into the source file.

Uses Allowed At Program Level Only

A uses directive has been coded inside a procedure or function which is not callable from the main program. This directive must occur in the main program block.

Wrong Number Of Selectors

A variant record has been defined, but each of the possible values for the selector has not been listed. Note that if you define a variant record such as

```
var allNumbers:  record
   case integer of
```

then you must list all of the possible integers, which, under ORCA/Pascal, would range from -maxint to +maxint (-32767 to 32767).

Zero String Not Allowed

Strict conformance to the ISO standard has been requested with the iso+ directive, and an attempt to assign null to a string has been found. Standard Pascal does not allow zero length strings.

Terminal Compilation Errors

A terminal error encountered during compilation will abort the compile. If you have compiled the program from the command line without using the -E switch, the compiler will enter the editor with the cursor on the offending line, and the error message will be displayed in the editor's information bar. If you have used the -E flag, the compiler will abort to the shell. If you compiled the program from an EXEC file, the default action is to display the error message and return to the shell.

Non-Pascal File Opened At An Inappropriate Time

Somewhere before the end of the program a source file written in a language other than Pascal was opened by means of the append directive. If you are chaining to a source file written in another language, the append directive should be placed after the end of the Pascal program.

Not Enough Bank Zero Memory

This error is usually caused by running a program prior to compilation of the current program, and the previous program has allocated a large amount of bank zero (direct page) memory without releasing it upon terminate. You must reboot the system in order to clean up memory. The error can also be caused by requesting more bank zero memory than is available.

Out Of Memory

There is not enough memory to compile the program. This can be caused by running programs which allocate memory and then fail to release it upon termination, or by lack of extended memory in your computer. ORCA/Pascal compiler requires 1.25M of memory. You can make more memory available by deallocating or reducing the size of any RAM disks you may have set up, and by temporarily removing disk caching programs and classic desk accessories.

User Termination

The user has entered the two-key abort command . (hold down the open-Apple key and then type a period). This does no harm to the program, it merely terminates compilation.

Linking Errors

Linker Error Levels

For each error that the linker can recover from, there is an error level which gives an indication of the gravity of the error. The table below lists the error levels and their meaning. Each error description shows the error level in brackets, immediately following the message. The highest error level found is printed at the end of the link edit. Many of these errors can only result if your program is written in more than one language, such as a combination of Pascal and assembly language. All linker errors are included here for completeness.

Severity	Meaning
2	Warning - things may be OK.
4	Error - an error was made, but the linker thinks it knows the programmer's intent and has corrected the mistake. Check the result carefully!
8	Error - no correction is possible, but the linker knew how much space to leave. A debugger can be used to fix the problem without recompiling.
16	Serious Error - it was not even possible to tell how much space to leave. Recompiling and linking will be required to fix the problem.

Recoverable Linker Errors

When the linker detects a nonfatal error, it prints

1. The name of the segment that contained the error.
2. How far into the segment (in bytes) the error point lies.
3. A text error message, with the error-level number in brackets immediately to the right of the message.

Addressing Error [16]

A label could not be placed at the same location on pass 2 as it was on pass 1.

This error is almost always accompanied by another error, which caused this one to occur; correcting the other error will correct this one. If there is no accompanying error, check for disk errors by doing a full assembly and link. If the error still occurs, report the problem as a bug.

Address Is Not In Current Bank [8]

The (most-significant-truncated) bytes of an expression did not evaluate to the value of the current location counter.

For short-address forms (6502-compatible), the truncated address bytes must match the current location counter. This restriction does not apply to long-form addresses (65816 native-mode addressing).

Address Is Not Zero Page [8]

The most significant bytes of the evaluated expression were not zero, but were required to be zero by the particular statement in which the expression was used.

This error occurs only when the statement requires a direct page address operand (range = 0 to 255).

Alignment Factor Must Be A Power Of Two [8]

An alignment factor that was not a power of 2 was used in the source code. In ORCA Assembly language, the ALIGN directive is used to set an alignment factor.

Alignment Factor Must Not Exceed Segment Align Factor [8]

An alignment factor specified inside the body of an object segment is greater than the alignment factor specified before the start of the segment. For example, if the segment is aligned to a page boundary (ALIGN = 256), you cannot align a portion of the segment to a larger boundary (such as ALIGN = 1024).

Code Exceeds Code Bank Size [4]

The load segment is larger than one memory bank (64K bytes). You have to divide your program into smaller load segments. See the description of the segment statement and MemoryModel directive for ways to do this.

Data Area Not Found [2]

A USING directive was issued in a segment from the ORCA/M assembler, and the linker could not find a DATA segment with the given name. Ensure that the proper libraries are included, or change the USING directive.

Duplicate Label [8]

A label was defined twice in the program. Remove one of the definitions.

Expression Operand Is Not In Same Segment [8]

An expression in the operand of an instruction or directive includes labels that are defined in two different relocatable segments. The linker cannot resolve the value of such an expression.

Evaluation Stack Overflow [8]

1. There may be a syntax error in the expression being evaluated.

Check to see if a syntax error has also occurred; if so, correct the problem that caused that error.

2. The expression may be too complex for the linker to evaluate.

Simplify the expression. An expression would have to be extremely complex to overflow the linker's evaluation stack, particularly if the expression passed the assembler or compiler without error.

Expression Syntax Error [16]

The format of an expression in the object module being linked was incorrect.

This error should occur only in the company of another error; correct that error and this one should be fixed automatically. If there are no accompanying errors, check for disk errors by doing a full assembly and link. If the error still occurs, report the problem as a bug.

Invalid Operation On Relocatable Expression [8]

The ORCA linker can resolve only certain expressions that contain labels that refer to relocatable segments. The following types of expressions *cannot* be used in an assembly-language operand involving one or more relocatable labels:

A bit-by-bit NOT
A bit-by-bit OR
A bit-by-bit EOR
A bit-by-bit AND
A logical NOT, OR, EOR, or AND
Any comparison (<, >, <>, <=, >=, ==)
Multiplication
Division
Integer remainder (MOD)

The following types of expressions involving a bit-shift operation *cannot* be used:

The number of bytes by which to shift a value is a relocatable label.
A relocatable label is shifted more than once.
A relocatable label is shifted and then added to another value.
You cannot use addition where both values being added are relocatable (you *can* add a constant to a relocatable value).
You cannot subtract a relocatable value from a constant (you *can* subtract a constant from a relocatable value).
You cannot subtract one relocatable value from another defined in a different segment (you *can* subtract two relocatable values defined in the same segment).

Only JSL Can Reference Dynamic Segment [8]

You referenced a dynamic segment in an instruction other than a JSL. Only a JSL can be used to reference a dynamic segment.

ORG Location Has Been Passed [16]

The linker encountered an ORG directive (created using the ORCA/M macro assembler) for a location it had already passed.

Move the segment to an earlier position in the program. This error applies only to absolute code, and should therefore be rarely encountered when writing for the Apple IIGS.

Relative Address Out Of Range [8]

The given destination address is too far from the current location.

Change the addressing mode or move the destination code closer.

Segment Header MEM Directive Not Allowed [16]

The linker does not support the MEM directive. If you are trying to use the MEM directive, you have probably made a mistake. MEM does not make sense in a relocatable load file.

Segment Header ORG Not Allowed [16]

If there is no ORG (created using the ORCA/M macro assembler) specified at the beginning of the source code, you cannot include an ORG within the program. The linker generates relocatable code unless it finds an ORG before the start of the first segment. Once some relocatable code has been generated, the linker cannot accept an ORG.

Shift Operator Is Not Allowed On JSL To Dynamic Segment [8]

The operand to a JSL includes the label of a dynamic segment that is acted on by a bit-shift operator. You probably typed the wrong character, or used the wrong label by mistake.

Undefined Opcode [16]

The linker encountered an instruction that it does not understand. There are four possible reasons:

1. The linker is an older version than that required by the assembler or compiler; in this case, a Linker Version Mismatch error should have occurred also. Update the linker.
2. An assembly or compilation error caused the generation of a bad object module. Check and remove all assembly/compilation errors.
3. The object module file has been physically damaged. Recompile to a fresh disk.
4. There is a bug in the assembler, compiler or linker. Please report the problem for correction.

Unresolved Reference [8]

The linker could not find a segment referenced by a label in the program.

If the label is listed in the global symbol table after the link, make sure the segment that references the label has issued a USING directive for the segment that contains the label. Otherwise, correct the problem by: (1) removing the label reference, (2) defining it as a global label, or (3) defining it in a data segment.

Multiple unresolved reference errors are generally caused by the libraries not being in the correct order. Use the command

```
COMPRESS A C 2/
```

to properly order the libraries. Commercial libraries supplied with compilers not developed by the Byte Works should not be included in the same library directory used by an ORCA language product.

Terminal Linker Errors

Could Not Open File *filename*

GS/OS could not open the file *filename*, which you specified in the command line.

Check the spelling of the file name you specified. Make sure the file is present on the disk and that the disk is not write-protected.

Could Not Overwrite Existing File *filename*

The linker is only allowed to replace an existing output file if the file type of the output file is one of the executable types. It is not allowed to overwrite a TXT, SRC, or OBJ file, thus protecting the unaware user.

Could Not Write The Keep File *filename*

A GS/OS error occurred while the linker was trying to write the output file *filename*.

This error is usually caused by a full disk. Otherwise, there may be a bad disk or disk drive.

Dictionary File Could Not Be Opened

The dictionary file is a temporary file on the work prefix that holds information destined for the load file's relocation dictionary. For some reason, this file could not be opened.

Use the SHOW PREFIX command to find out what the work prefix is. Perhaps you have assigned the work prefix to a RAM disk, but do not have a RAM disk on line. Have you removed the volume containing the work prefix from the disk drive? Is the disk write-protected?

Expression Recursion Level Exceeded

It is possible for an expression to be an expression itself; therefore, the expression evaluator in the linker is recursive. Generally, this error occurs when the recursion nest level exceeds ten. This should not happen very frequently. If it does, check for expressions with circular definitions, or reduce the nesting of expressions.

File Read Error

An I/O error occurred when the linker tried to read a file that was already open. This error should never occur. There may be a problem with the disk drive or with the file. You might have removed the disk before the link was complete.

File Not Found *filename*

The file *filename* could not be found.

Check the spelling of the file name in both the KEEP directive and the LINK command. Make sure the .ROOT or .A file has the same prefix as the file specified in those commands.

Illegal Header Value

The linker checks the segment headers in object files to make sure they make sense. This error means that the linker has found a problem with a segment header.

This error should not occur. Your file may have been corrupted, or the assembler or compiler may have made an error.

Illegal Segment Structure

There is something wrong with an object segment.

This error should not occur. Your file may have been corrupted, or the assembler or compiler may have made an error. This can also be caused by a bad disk or bad memory chip. Try changing to a different disk and recompiling.

Invalid File Name *filename*

The file *filename* does not adhere to GS/OS file naming conventions.

Make sure the file name you supply on the command line is a valid one.

Invalid file type *filename*

The file *filename* is not an object file or library file.

Check the shell command line to make sure you didn't list any files that are not object files or library files. Check your disk directory to make sure there isn’t a non-object file with the same root name as a file you are linking. For example, if you are linking object files name

MYFILE.ROOT and MYFILE.A, make sure there is no (unrelated) file on the disk with the name MYFILE.B.

Invalid Keep Type

The linker can generate several kinds of output files. The type of the output file must be one of the executable types. Since it is possible to set the keep type with a shell variable, this error can occur from a command line call.

Linker Version Mismatch

The object module format version of the object segment is more recent than the version of the linker you are using.

Check with the Byte Works to get the latest version of ORCA.

Must Be An Object File *filename*

Filename is not an object file or a library file.

Object Module Read Error

A GS/OS error occurred while the linker was trying to read from the currently opened object module.

This error may occur after a nonfatal error; correcting the nonfatal errors may correct this one. Otherwise, it may be caused by a bad disk or disk drive.

Out Of Memory

All free memory has been used; the memory needed by the linker is not available.

Execution Errors

Each of the following errors can be trapped by the built-in procedure SystemError. The error numbers returned by SystemError are printed after the error message in the descriptions below. Each error description explains the action that will be taken by the system. You can use this to decide how to handle a particular error.

Subrange Exceeded [1]

A value has been assigned to a variable, and the value exceeds the bounds defined for the variable. This error can generally be detected during compilation by enabling the RangeCheck directive. For variables of type byte, the value was not within the range 0..255. For boolean variables, the value was neither a 0 nor a 1. For char variables, the value was not within 0..127. This sort of error occurring for other types of variables will be flagged by different error messages.

The results of the assignment, and thus whether execution should continue, depend upon the variable type. If the variable is an element of a char or boolean array that is being indirectly accessed, as by assignment to a var parameter, the value will be stored into the least significant byte of the element. If the array is being accessed directly, then an entire word will be stored into the element.

Use of char or boolean values beyond their defined range will produce unpredictable results.

File Is Not Open [2]

A file operation has been coded, such as a `read(filePointer)` or `write(filePointer)`, but the requested file has not been opened with an open, reset, or rewrite call, or has been closed with a `close` call. The action taken by the system is to not perform the operation; the file variable value is undefined.

Read While At End Of File [3]

The file pointer attached to a particular file is pointing to the end-of-file marker. The area pointed to by the input buffer pointer is unpredictable.

I/O Error [4]

A ProDOS I/O error has been detected, such as disk full, write error, etc., or an illegal file operation has been coded. The system ignores the operation.

Out Of Memory [5]

An allocation of memory using the built-in procedure new was attempted, but no free memory is available. The system will set the pointer to the memory to nil.

EOLN While At End Of File [6]

In Pascal, input files always have an end-of-line marker preceding the end-of-file marker. The program failed to check for EOF after detecting an EOLN. If input is coming from a file, EOF will remain true. If input is coming from the keyboard, EOF will be set to false.

Set Overflow [7]

An operation on a set caused the set to exceed the size allocated to it. All set overflows will be detected if range checking is enabled. You should be aware that some set overflows will not be caught when range checking is off, however. If a set occupies an exact amount of bytes, then an element out of range will always be detected. If a set does not occupy an exact number of bytes, then the compiler may fail to find an illegal assignment with range checking off. This could occur, for example, if the set had been defined as `var s1: set of 0..10;` and then `s1` was assigned an 11. `S1` occupies two bytes, and 11 falls within the bits marked for assignment to `s1`.

If you trap this error and return control to the executing program, the extra bytes in the set value are truncated.

Jump To Undefined Case Statement Label [8]

This error cannot be recovered from. Your program can avoid this error by including an otherwise clause in the case statement.

Integer Math Error [9]

An overflow has occurred as a result of an addition, subtraction, or multiplication, or an attempt has been made to divide by zero, in either a division or div operation. The action taken by the system depends upon the operation, but the result is neither valid nor predictable.

Real Math Error [10]

An exception has been raised due to one of the following invalid conditions:

1. Addition or subtraction of values whose magnitudes are considered infinities.
2. Multiplication by infinity.
3. Division by zero, or division of infinity by infinity.
4. Taking the square root of a negative value.
5. Conversion of a real number to an integer value such that the real number is of a magnitude considered to be infinity.
6. Comparison of two values, using the operators < or >, where one of the values is considered to be NaN (not a number).
7. Any operation involving an NaN.

The system returns NaN as the result of the operation.

Underflow [11]

A value is too close to zero to be represented in the accuracy of the underlying format. The system returns a value of zero.

Overflow [12]

A value is too large to be represented in the accuracy of the underlying format. The system returns a value of infinity (INF).

Divide By Zero [13]

An attempt has been made to divide by zero. The system returns NaN (not a number).

Inexact [14]

An attempt has been made to perform an otherwise legitimate trigonometric function, such as sin(x), with numbers of excessive magnitude. The system returns its best guess at the true answer.

Stack Overflow [15]

When range checking is enabled, code is generated to check to see if there is enough room on the stack to form a new stack frame when a procedure or function is called. This error is flagged if there is not enough room. The stack has not yet been damaged, so it is safe to simply exit the program.

Stack Error [16]

The stack has overflowed, overwriting memory that does not belong to your program. This could cause crashes or other odd behavior. The best course of action upon detecting this error is to call SysFailMgr in the Miscellaneous Tool Set to force a reboot.

Appendix C – Custom Installations

This appendix is designed to help you install ORCA/Pascal to take advantage of your specific hardware configuration. As shipped, ORCA/Pascal is set up for people who have one or two 3.5 inch floppy disk drives and who want to use the desktop programming environment. If that describes your system, you should make copies of the original disks, and use them just as they were shipped. If you have a hard disk or prefer a text-based programming environment, you can use Apple's Installer program to create an ORCA/Pascal environment that suits your needs. Finally, this appendix describes the principal files that make up the ORCA development environment and the Pascal compiler; by studying this section, you can learn why we configured ORCA/Pascal the way we did, and adjust the installation to suite your needs.

Installer Scripts

Apple's Installer can be used to create a floppy-disk based text programming environment or to install ORCA/Pascal on your hard disk, either as a separate language or in combination with other ORCA languages. To run the installer, execute the Installer file from the ORCA/Pascal Extras disk. There are several installer scripts listed in the window that appears; these are described below. Select the one you want, select the disk that you want to install the program on from the right-hand list, and click on the Install button.

Please note that with the current version of Apple's Installer, you will have to select the installation script before you can pick a folder from the right-hand list.

Low Memory

This installation script should be used after one of the other scripts that installs ORCA/Pascal. It installs a smaller version of the ORCA/Pascal compiler. The smaller compiler ignores the optimize directive. Individual subroutines are also limited to 16K rather than 64K. The small memory compiler can be used when memory is tight, though, and should always be used if PRIZM is being used on any system with less than 1.75M.

New System

This is the basic, all-purpose installation script. It installs the full ORCA/Pascal system, including the desktop development system, the text based editor, and all of the interface files and help files that you don't have enough room for from a floppy-disk based system. You will need a hard disk to install the full ORCA/Pascal system. In addition, you should have at least 1.75M of memory; if not, see the "Low Memory" script, described below.

If you run a lot of software, you probably boot into the Finder or some other program launcher. In that case, you should probably install ORCA/Pascal in a folder that is not at the root level of your hard disk.

If you plan to use your computer primarily for programming, you can set things up so you boot directly into ORCA/Pascal. To do that, start by installing Apple's system disk without a Finder. (Apple's installer, shipped on their system disk and available free from your local Apple dealer, has an installation option to install the system with no Finder.) Next, install ORCA/Pascal at the root level of your boot volume, making sure that ORCA.Sys16 is the first system file on the boot disk. System files are those files with a file type of S16 that end with the characters ".Sys16", as well as the files with a file type of SYS that end in the characters ".SYSTEM".

See also "ORCA Icons" and "ORCA Pascal, C, Asm Libraries," below.

New Text System

This script is ideal if you plan to do "standard" Pascal programming, and prefer a UNIX-like text environment. The text system is small enough to fit on a single 800K floppy disk, although you can install it on a hard disk as well. The desktop development system is not installed. Tool header files are not installed, but you can copy any you plan to use manually. You can use any copy program to copy the interface files, which are located in the folder "Libraries/ORCAPascalDefs" on the ORCA.Pascal disk and the Extras disk. Copy any of them that you want to the same folder wherever you installed ORCA/Pascal.

If you run a lot of software, you probably boot into the Finder or some other program launcher. In that case, you should probably install ORCA/Pascal in a folder that is not at the root level of your hard disk.

If you plan to use your computer primarily for programming, you can set things up so you boot directly into ORCA/Pascal. To do that, start by installing Apple's system disk without a Finder. (Apple's installer, shipped on their system disk and available free from your local Apple dealer, has an installation option to install the system with no Finder.) Next, install ORCA/Pascal at the root level of your boot volume, making sure that ORCA.Sys16 is the first system file on the boot disk. System files are those files with a file type of S16 that end with the characters ".Sys16", as well as the files with a file type of SYS that end in the characters ".SYSTEM".

See also "ORCA Icons" and "ORCA Pascal, C, Asm Libraries," below.

ORCA Icons

If you use Apple's Finder as a program launcher, be sure and install the ORCA Icons. ORCA itself will show up as a whale, while the various source files, object files, and utilities will be displayed with distinctive icons.

ORCA Pascal, C, Asm Libraries

If you are using ORCA/Pascal with the ORCA/M macro assembler, installing ORCA/Pascal gives you all of the libraries you need.

If you are using ORCA/C and ORCA/Pascal together on the same system, you must have a total of four library files in your library folder, and they must appear in the correct order. If you are missing any of the libraries, or if they are in the wrong order, you will get linker errors with either C, Pascal, or possibly with both languages. This installer script installs the libraries for C, Pascal, and assembly language in the correct order. (The libraries used by the assembler are also used by C and Pascal, so you get them anytime you use C or Pascal.) You can use this installer script before or after any of the other scripts.

You should not use this script unless you are installing C and Pascal together. Installing the Pascal libraries takes up a little more room on your disk; slows link times a little, since the linker has to scan an extra library; and uses up a little extra memory, since the library header is loaded by the linker.

Update System

This script will update an old Pascal system or add Pascal to an existing ORCA/M or ORCA/Pascal system. All of the executable files from the Pascal disk are copied to your old system, but the LOGIN file, SYSCMND file, SYSEMAC file, SYSTABS file and SYSHELP file are not updated, since all of these may have been customized in your old system. Of course, if you are installing Pascal into an existing system that does not already have Pascal, you will need to add the Pascal language to your SYSCMND file. If you are installing Pascal in an existing Pascal system, be sure and follow up this step with the "ORCA Pascal, C, Asm Libraries" script, described above.

This installation option should not be used to update a Pascal 1.0 system to Pascal 2.0. It can be used to update any 2.0 level ORCA installation, but you should install ORCA/Pascal in a new folder if your current version is prior to 2.0.

See also "ORCA Icons" and "ORCA Pascal, C, Asm Libraries."

Update Text System

This script will update an old Pascal system or add Pascal to an existing ORCA/M or ORCA/Pascal system. This script does not install or update the PRIZM desktop development system, nor does it install the interface files for the Apple IIGS toolbox, ProDOS, GS/OS or the ORCA shell. All of the executable files except those used only for the PRIZM desktop development system are copied to your old system, but the LOGIN file, SYSCMND file, SYSEMAC file, SYSTABS file and SYSHELP file are not updated, since all of these may have been customized in your old system. Of course, if you are installing Pascal into an existing system that does not already have Pascal, you will need to add the Pascal language to your SYSCMND file. If you are installing Pascal in an existing Pascal system, be sure and follow up this step with the "ORCA Pascal, C, Asm Libraries" script, described above.

While the interface files are not installed, you can copy any of them that you would like to use after updating the basic text system. You can use any copy program to copy the headers, which are located in the folder "Libraries/ORCAPascalDefs" on the ORCA.Pascal disk and the Extras disk. Copy any of them that you want to the same folder wherever you installed ORCA/Pascal.

This installation option should not be used to update a Pascal 1.0 system to Pascal 2.0. It can be used to update any 2.0 level ORCA installation, but you should install ORCA/Pascal in a new folder if your current version is prior to 2.0.

See also "ORCA Icons" and "ORCA Pascal, C, Asm Libraries."

Update Text System, No Editor

This installation script is basically the same as the "Update Text System" script, described above, but it doesn't install the text editor. It may seem silly at first to install a text system with no text editor, but there are a number of text-based editors available from third party sources; this installation option installs Pascal without removing your existing text editor.

See also "ORCA Icons" and "ORCA Pascal, C, Asm Libraries."

RAM Disks

RAM disks come in a variety of sizes and flavors. One of the most common is a RAM disk allocated from the control panel of your computer.

△ **Important** We do not recommend using a RAM disk of this kind unless you have only one 3.5" floppy disk, and then we recommend keeping it small and using it only for temporary storage. △

These RAM disks are allocated from the memory of your computer. ORCA/Pascal can make very effective use of that memory if you let it – the system will perform better than if you try to copy parts of ORCA to your RAM disk. In addition, RAM disks allocated from main memory are easy to destroy from a program that is accidentally writing to memory that it has not allocated. While this is unusual in commercial programs, you may find that your own programs do this frequently during the development cycle. RAM disks that are not allocated from main memory, like Apple's "Slinky" RAM disk, are good for work space and even source code. The so-called ROM disks, or battery-backed RAM disks, should be treated as small hard disks. See the sections on installing ORCA/Pascal on a hard disk for effective ways of using ROM disks.

Details About Configuration

In this section, we will explore why ORCA/Pascal is configured the way it is by looking at what happens when you run ORCA/Pascal, when ORCA looks for files, and where it looks for files. The material in this section is advanced information for experienced programmers. You do not need to understand this material for beginning and intermediate programming, and the entire section can safely be skipped.

Whether you are using the text or desktop programming system, you always start ORCA/Pascal by running the ORCA.Sys16 file. This file contains the UNIX-like text based shell. The first thing the shell does after starting is to look for a folder called Shell; this folder must be in the same location as the ORCA.Sys16 file. Inside this folder, the shell looks for an ASCII file (it can be stamped as a ProDOS TXT file or an ORCA SRC file) with the name SYSCMND; this is the command table. It is loaded one time, and never examined again. The shell must get at least this far, successfully loading the SYSCMND table, or it will stop with a system error.

The next step taken by the shell is to set up the default prefixes. Prefix 8 is not changed if it has already been set by the program launcher, but the shell will set it to the same prefix as prefix 9 if prefix 8 is initially empty. The remaining prefixes default to prefix 9 plus some subdirectory, as show in the table below.

prefix	set to
`13`	`9:libraries`
`14`	`9`
`15`	`9:system`
`16`	`9:languages`
`17`	`9:utilities`

The last step taken by the shell is to look in prefix 15 for a script file named LOGIN. To qualify, this file must have a file type of SRC, and a language stamp of EXEC. If the shell does not find a valid LOGIN file, it simply moves on; in other words, you can leave out the LOGIN file if you choose. Typically, this script file is used to set up custom aliases, set up shell variables, change the default prefixes listed above to other locations, and to execute PRIZM, the desktop development system. One thing this shows is that, as far as ORCA is concerned, the PRIZM desktop development system is actually nothing more than an application that you run from within the shell. Systems that default to the desktop programming environment do so by running PRIZM from within the LOGIN script, so PRIZM is executed as part of the boot process.

After executing the LOGIN script, the shell writes a # character to the screen and waits for further commands. If course, if PRIZM is executed from the LOGIN file, the shell never gets a chance to do this until you quit from PRIZM.

Prefixes 13 to 17 are initialized by the shell, but you can change them to point to other folders if you prefer. To understand how these prefixes are used, we'll look at the programs that currently use them.

When you use the EDIT command, the shell attempts to run a program named EDITOR; it expects to find an EXE file with this name in prefix 15 (the "Shell" prefix). If the shell does not find an EXE file with the name EDITOR in prefix 15, it writes the message "ProDOS: File not found" and returns to the # prompt. The ORCA editor uses prefix 15 to locate the SYSTABS file (to set up the tab line), the SYSEMAC file (to set up the default editor macros), and the SYSHELP file (to write the editor help screen). The editor can function perfectly well without any of these files, although you will get a warning message each time you load a file if there is no SYSTABS file. When you cut, copy or paste text, the editor reads or writes a file called

SYSTEMP to prefix 14; obviously, the editor will perform a lot faster on these operations if prefix 14 is set to point to a RAM disk.

A few other programs look at the SYSTABS file in prefix 15; PRIZM is another good example. No other use is currently made of prefix 15.

Prefix 14, which the editor uses as a work prefix, is also used by the shell when you pipe output from one program to become input to another program. The shell handles piping by creating a temporary file to hold the output of one program, reading this file as standard input for the next program. These pipe files are called SYSPIPE0, SYSPIPE1, and so forth, depending on how many pipes were used on a single command line.

When you use any of the commands to compile or link a program, the shell looks in prefix 16 for the compiler, assembler, or linker. For example, if you compile a Pascal source file, the shall takes a look at the auxtype field for the file, which will have a value of 5. The shell then scans its internal copy of the SYSCMND file looking for a language with a number of 5, and finds one with a name of PASCAL. The shell then loads and executes the file 16:PASCAL; if it does not find such a file, it flags a language not available error.

Compilers and linkers make heavy use of prefix 13, which is not actually used by the shell. Prefix 13, the library prefix, is where the Pascal compiler looks for interface files. When you code a uses statement like this one:

```
uses QuickDrawII;
```

the Pascal compiler appends this file name to the prefix 13:ORCAPascalDefs, giving a full path name for the file of 13:ORCAPascalDefs:QuickDrawII.int. The ORCA/C compiler does something similar, but it uses 13:ORCACDefs. A convention has also gradually developed to put assembler macros and equate files in a folder called AInclude or ORCAInclude inside the library folder, although the assembler and MACGEN utility don't automatically scan this folder.

The linker also uses the library folder. When you link a program, especially one written in a high-level language, the program almost always needs a few subroutines from a standard library. The linker recognizes this automatically, and scans prefix 13 looking for library files. The linker ignores any folders or other non-library files it might find. When the linker finds a library file, it opens it, scans the files in the library to resolve any subroutines, closes the file, and moves on. The linker never goes back to rescan a library, which is why it is important for the libraries to be in the correct order.

Prefix 17 is the utility prefix. When you type a command from the shell, the shell checks to see if it is in the command table. If so, and if the file is a utility, the shell appends the name to 17: and executes the resulting file. For example, when you run the MAKELIB utility to create your own Pascal library, the shell actually executes the file 17:MAKELIB, giving a file not found error if there is no such file. Utilities are not limited to EXE type files; you can make an SYS file, S16 file, or script file a utility, too.

Prefix 17 is also used by the help command. When you type HELP with no parameters, the help command dumps the command names from the SYSCMND table. When you type HELP with some parameter, like

```
help catalog
```

the help command looks for a text (TXT) or source (SRC) file named 17:HELP:CATALOG, typing the file if it is found. In other words, you can use the help command to type any standard file, as long as you put that file in the HELP folder inside of the utilities folder.

All of the files that were not mentioned in this section can be placed absolutely anywhere you want to put them – since none of the ORCA software looks for the files in a specific location, you have to tell the system where they are anyway. It might as well be a location you can remember, so pick one that makes sense to you.

All of this information can be put to use for a variety of purposes. For example, by installing the Finder, BASIC.SYSTEM, and any other programs you use regularly as utilities under ORCA, you can boot directly into ORCA's text environment (which takes less time than booting into the Finder) and use ORCA as a program launcher. You can also split the ORCA system across several 3.5" floppy disks by moving, say, the libraries folder to the second disk, setting prefix 13 to point to the new disk from within your LOGIN file.

Appendix D – Run-Time License

Any program written with ORCA/Pascal has some of Pascal's run-time libraries linked into the executable program. These libraries are copyrighted by the Byte Works. While we feel you should be able to use these libraries free of charge for any program you write, commercial or not, there are also a few legal requirements that must be met so we can protect our copyright.

On any program written with ORCA/Pascal, you must include a copyright statement stating that the program contains copyrighted libraries from the ORCA/Pascal run-time library. This copyright must appear in a prominent place. If the program has any other copyright statement, the ORCA/Pascal copyright statement must appear in the same location. The text that must be included is:

> This program contains material from the ORCA/Pascal
> Run-Time Libraries, copyright 1987-1993
> by Byte Works, Inc. Used with permission.

Appendix E – Console Control Codes

When you are writing programs that will be executed in a text environment, you can use a number of special console control codes. These are special characters which cause the console to take some action, like moving the cursor or turning the cursor off. This appendix gives a list of the most commonly used console control codes for the GS/OS .CONSOLE driver; this is the default text output device used by ORCA/Pascal for stand-alone text programs and for programs executed under the text shell.

If you send output to some output device other than the GS/OS .CONSOLE driver that ORCA/Pascal 2.0 uses as the default output device, some of these control codes may not be respected, and others may be available. In general, very few console control codes are recognized when output is sent to a graphics device or printer.

For a complete list of the console control codes supported by the GS/OS .CONSOLE driver, see Apple IIGS GS/OS Reference, pages 242-245.

Beep the Speaker

Writing chr(7) to the screen beeps the speaker.

```
procedure Beep;

{ Beep the speaker                                          }

begin {Beep}
write(chr(7));
end; {Beep}
```

Cursor Control

There are several control codes that move the cursor. Chr(8) moves the cursor one column to the left. If the cursor starts in column 0 (the leftmost column), it is moved to column 79 (the rightmost column) of the line above. If the cursor starts in column 0 of row 0 (the top row), it is moved to column 79 of row 0.

Chr(10) moves the cursor down one line. If the cursor starts on line 23 (the bottom line on the screen), the screen scrolls up one line and the cursor stays in the same relative position.

Chr(28) moves the cursor right one column. If the cursor starts in column 79, it is moved to column 0 of the same line. This curious behavior is worth noting: you would normally expect that the cursor would move to column 0 of the *next* line, not the current line, especially when the action of chr(8) is taken into account.

Chr(31) moves the cursor up one line. If the cursor starts on line 0, nothing happens.

Chr(25) moves the cursor to line 0, column 0, which is the top left character on the screen. It does not clear the screen; it simply moves the cursor. Chr(12) also moves the cursor to the top left of the screen, but is also clears the screen to all blanks. Note that writing chr(12) is equivalent to using the Standard Pascal procedure Page. In fact, the Page procedure writes chr(12) to the output device.

Chr(13) moves the cursor to the start of the current line.

Chr(30) is the only control code that requires more than one character. This character starts a cursor movement sequence which depends on the two characters that follow. Using this character code, the cursor can be positioned to any column and row on the screen. The first character after the chr(30) is used to position the cursor in a particular column. The column number is computed by subtracting 32 from the ordinal value of the character written. The next character determines the row, also by subtracting 32 from the ordinal value of the character. For example,

```
write(chr(30), chr(10+32), chr(5+32));
```

would move the cursor to column 10, row 5. Columns and rows both start with number 0, so that the upper-left screen position is at row 0, column 0, and the lower-right screen position is at row 23, column 79. See the GotoXY procedure, below, for a convenient way of using this feature.

```
procedure FormFeed;

{ Move the cursor to the top left of the screen and clear the  }
{ screen                                                       }

begin {FormFeed}
write(chr(12));
end; {FormFeed}


procedure GotoXY (x, y: integer);

{ Move the cursor to column x, row y                           }

begin {GotoXY}
write(chr(30), chr(x+32), chr(y+32));
end; {GotoXY}


procedure Home;

{ Move the cursor to the top left of the screen                }

begin {Home}
write(chr(25));
end; {Home}
```

```
procedure MoveDown;

{ Move the cursor down                                          }

begin {MoveDown}
write(chr(10));
end; {MoveDown}


procedure MoveLeft;

{ Move the cursor to the left                                   }

begin {MoveLeft}
write(chr(8));
end; {MoveLeft}


procedure MoveRight;

{ Move the cursor to the right                                  }

begin {MoveRight}
write(chr(28));
end; {MoveRight}


procedure MoveUp;

{ Move the cursor up one line                                   }

begin {MoveUp}
write(chr(31));
end; {MoveUp}


procedure Return;

{ Move the cursor to the start of the current line              }

begin {Return}
write(chr(13));
end; {Return}
```

Clearing the Screen

In the last section, we looked at chr(12), which clears the screen and moves the cursor to the top-left of the screen. There are two other control codes that can clear parts of the screen. Chr(11) clears the screen from the cursor position to the end of the screen, filling the cleared area with blanks. Chr(29) is still more selective. It clears the screen from the current character position to the end of the line.

```
procedure ClearToEOL;

{ Clear to the end of the current line                         }

begin {ClearToEOL}
write(chr(29));
end; {ClearToEOL}


procedure ClearToEOS;

{ Clear to the end of the screen                               }

begin {ClearToEOS}
write(chr(11));
end; {ClearToEOS}
```

Inverse Characters

Text normally shows up on the text screen as white characters on a black background. Writing chr(15) causes any future characters to be written as black characters on a white background. Chr(14) reverses the effect, writing white characters on a black background.

```
procedure Inverse;

{ Write all future characters in inverse                       }

begin {Inverse}
write(chr(15));
end; {Inverse}


procedure Normal;

{ Write future characters in normal mode (white on black)      }

begin {Normal}
write(chr(14));
end; {Normal}
```

MouseText

The Apple IIGS text screen has a set of graphics characters called MouseText characters. The name comes from the primary purpose for the characters, which is to implement text-based desktop environments for use with a mouse, like the text version of Apple Works. You need to do two things to enable MouseText characters: enable MouseText, and switch to inverse characters. After taking these steps, any of the characters from '@' to '_' in the ASCII character set will show up as one of the graphics characters from the MouseText character set. Chr(27) turns MouseText on, while chr(24) turns it off.

You need to be sure and turn MouseText off if you turn it on – the ORCA shell expects to me in normal mode when your program is finished.

```
procedure MouseTextOn;

{ Turn mousetext on                                           }

begin {MouseTextOn}
write(chr(27));
end; {MouseTextOn}


procedure MouseTextOff;

{ Turn mousetext off                                          }

begin {MouseTextOff}
write(chr(24));
end; {MouseTextOff}
```

The best way to see the MouseText characters and to see which key each graphics character is associated with is with a short program. The example below assumes that you have typed in the inverse, normal, mousetexton and mousetextoff functions from this appendix; you need to put them right after the program statement.

```
program MouseText(output);

var
   ch: char

begin
for ch := '@' to '`' do
   write(ch, ' ');
writeln;
writeln;
MouseTextOn;
```

```
for ch := '@' to '`' do begin
   Inverse;
   write(ch);
   Normal;
   write(' ');
   end;
MouseTextOff;
writeln;
end.
```