ORCA/C 2.0™

A C Compiler
and Development System
for the
Apple IIGS

Mike Westerfield

Byte Works®, Inc.
8000 Wagon Mound Dr. NW
Albuquerque, NM 87120
(505) 898-8183

Credits

C Compiler
Mike Westerfield

Development Environment
Mike Westerfield
Phil Montoya

Testing
Barbara Allred
Mike Westerfield

Documentation
Mike Westerfield
Barbara Allred
Patty Westerfield

Table of Contents

Chapter 1 – Introducing ORCA/C

ORCA/C

Welcome to ORCA/C! ORCA/C 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 C compiler, a linker that lets you create and use libraries, or even mix C 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 C programs, showing you what line is executing and what the values of the variables are. 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/C, since the best approach is different for different people.

What You Need

To use ORCA/C, 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/C, 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/C. 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 disks, or if you have two 3.5" floppies and would like to use ORCA/C 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/C in a variety of different configurations; these are explained in detail in Appendix B. While these installation instructions make it easy to install ORCA/C, 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/C; if you find that you are confused, you might try using the floppy-disk based system until you learn how to use ORCA/C.

About the Manual

This manual is your guide to ORCA/C. 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 C 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 C programs. Part three is the "Language Reference Manual." It contains information about the ORCA/C 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 C programming language.

While this manual will teach you how to use ORCA/C to write and test programs, it does not teach you the basics of the language itself. Basic concepts about programming in C are necessary to create useful, efficient programs. If you are new to C, you can start with our Learn to Program in C course, which is written specifically for ORCA/C 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 C programs. The first four chapters give you enough information to create, test and debug C programs using ORCA/C. 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/C. 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.

Other Books and Reference Materials

If you are new to C, you will need to supplement this manual with a good beginner's book on the C programming language. A companion course is available from the Byte Works that teaches you the C language and some basic techniques for programming. The book is called Learn to Program in C, and it has one distinct advantage over any other C programming book: it is written specifically for ORCA/C running on an Apple IIGS. Another good way to find a book that suits you, especially if you already know a little C, 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 C; 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 C, which is a complete introduction to the world of toolbox programming.

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

Toolbox Programming in C
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.

<u>C: A Reference Manual, Third Edition</u>
Samuel P. Harbison and Guy L. Steele Jr.
Tartan Laboratories, Prentice-Hall, Inc., Englewood Cliffs, New Jersey
The definitive C reference manual.

<u>The C Programming Language, Second Edition</u>
Brian W. Kerninghan and Dennis M. Ritchie
Prentice-Hall Software Series, Englewood Cliffs, New Jersey
The original C "Bible." While this book is still a popular reference for the C language, we feel <u>C: A Reference Manual</u> does a better job of covering the language.

<u>C Primer Plus</u>
Mitchell Wait, Stephen Prata, Donald Martin
Howard W. Sams & Co., Inc., Indianapolis, Indiana
An excellent primer for beginners.

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

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

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

<u>Apple IIGS Toolbox Reference: Volume I, Apple IIGS Toolbox Reference: Volume II and Apple IIGS Toolbox Reference: Volume III</u>
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.;

<u>Programmer's Reference for System 6.0</u>
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 C programming, since C has library functions 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/C. Without changing programming environments, you can create a program in C, 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/Pascal
Mike Westerfield
Byte Works, Inc., Albuquerque, NM
ORCA/Pascal is a Pascal compiler which can be installed in the same environment as ORCA/C. With the Pascal compiler installed, you can write C or Pascal programs without switching environments. You can also use library routines written in Pascal from your C programs.

Chapter 2 – Exploring the System

Backing Up ORCA/C

This chapter is a hands-on introduction to ORCA/C. 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 C programming is like using ORCA/C. 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/C 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/C does write some information to disk as it prepares your programs. The first step is to start ORCA/C. 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/C.

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.cc. This file is the source code for the C 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/C 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/C. 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/C. 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 C 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.cc source file (a window is selected by clicking anywhere on the window) and pull down the Languages menu, you will see CC 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 C 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 C, not all languages that work with the development system support the source-level debugger. If you are using another language with ORCA/C, 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 to find problems in your C 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.

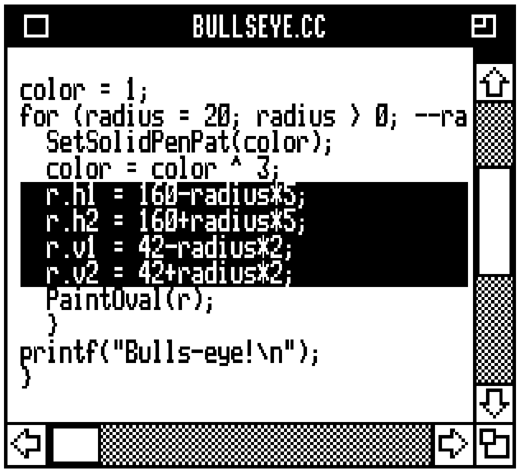
The Go Command

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.



```
color = 1;
for (radius = 20; radius > 0; --ra
  SetSolidPenPat(color);
  color = color ^ 3;
  r.h1 = 160-radius*5;
  r.h2 = 160+radius*5;
  r.v1 = 42-radius*2;
  r.v2 = 42+radius*2;
  PaintOval(r);
  }
printf("Bulls-eye!\n");
}
```

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.cc. 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.

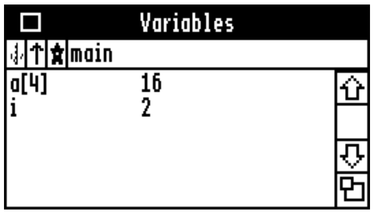
To see how the Step Through command works, let's debug the Sort.cc program. If you pull down the Debug menu, you will see that the Step Through, Go to Next Return, and Stop items are all dimmed, meaning that they are not selectable at this time. This is because there is nothing to step through or stop,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

Pull down the Debug menu and select the Step command. Sort.cc 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



```
a[4]        16
i           2
```

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.

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, structures 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 function. The name ShellSort appears in the information bar of the Variables window, the up-arrow is now selectable, and the variable a[4] vanishes. If you click on the up arrow to the left of the ShellSort function name, you will see the variables display for the main program, and the down arrow in the Variables window becomes selectable. 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 structure or union, 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 structure.

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. When dereferencing pointers, you need to use the Pascal pointer operator, which is a ^ character after the pointer, rather than C’s * character before the pointer. You can use the -> operator to look at a field in a structure that is pointed to by a pointer, just like you do in C. Finally, you can use pointer operators (^ and ->), field operators (the . character) and array subscripts in combination.

All variable names are case insensitive – that is, Size and SiZE are regarded as the same name. If you define two variables in the same function with names that differ only in the case of the letters, only one can be seen from the debugger, and there is no good way to predict which of the two will be visible.

Those Other Disks

There are three other disks in your ORCA/C 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/C 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/C or to install ORCA/C on a hard disk. Installation is covered in Appendix B. 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 email 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 chocked full of actual C 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 C, most books cover relatively straight-forward text-based programs that can be written using standard C. 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/C 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/C, 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/C follow a common convention. They look like a standard C preprocessor command. In all cases, ORCA/C's compiler directives start with the word pragma, which is used in C preprocessors for directives that are unique to a given compiler. That way, other C compiler will ignore the directives that are only used by ORCA/C, and ORCA/C will ignore directives for any other compiler. Most of the directives can be used anywhere in the source file.

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

```
#pragma keep "MYPROG"
#pragma memorymodel 1
```

A Brief Summary of ORCA/C Pragmas

The various pragmas are explained in detail later in this book. All of the pragmas are described in Chapter 12, and many are explained in the next chapter, which outlines the various kinds of programs you can write with ORCA/C. The table you see below gives you a brief overview of the pragmas so you know what is available.

pragma	use
cda	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.
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.
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. C 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/C to set the databank register properly, something it normally does not need to do.
debug	ORCA/C 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.
expand	The C preprocessor actually changes the program the C compiler compiles. This directive tells ORCA/C to write the output of the C preprocessor so you can see what the compiler it actually compiling; it is very handy when you are debugging preprocessor macros.
float	ORCA/C 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.
ignore	The C language has changed over the years, and one of the changes that helps a lot is that error checking is more extensive than it used to be. While extra error checking is generally a good thing, it can make it difficult to compile some outdated, otherwise correct, programs. This pragma lets you tell ORCA/C to ignore some kinds of errors, making it easier to port these programs.
keep	The keep pragma lets you hard code a specific keep name in the source file for your program.
lint	A lot of the safety associated with high-level languages is not present in the C language. This directive adds extra restrictions to the C language to catch some kinds of programming mistakes that the compiler generally ignores.
memorymodel	All 65816 programs must consist of chunks of code that are 64K or smaller, and ORCA/C further assumes that all arrays are structures 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 structures

	allocated using dynamic memory are smaller than 64K. The segment statement can also be used to break a program into smaller pieces.
nba	Used to create New Button Actions (NBAs) for HyperStudio. NBAs are discussed in Chapter 4.
nda	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.
noroot	When ORCA/C creates a program, it actually generates two files that are used later by a part of the development system called the linker. One of these, called the root file, contains startup code that must be executed to set up the program. In programs that consist of more than one separately compiled source file, you can use the noroot pragma in all of the source files except the one with main; this tells the compiler not to generate startup code for the source files that don't need it, making your program a little smaller.
optimize	ORCA/C 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 pragma lets you control the level of optimization.
path	When you use an include or append statement, you can tell ORCA/C to search the ORCACDefs folder or the current folder. In some cases, though, you may want ORCA/C to look other places for additional source files and header files – say, on a network computer that is also used by other programmers. The path pragma adds path names to the list of locations ORCA/C will look for a file.
rtl	ORCA/C 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.
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.
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 C does, though. This directive is used to tell the C compiler to create a function that returns values using the toolbox convention. You will also need to use the databank pragma, and it's generally a good idea to use the pascal directive to switch the order of the parameters, too.

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/C 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/C 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 simply include the appropriate standard C library interfaces and use any of the standard C output routines, like printf. Input is just as easy – you use standard C library routines like gets to read characters from the keyboard. (Standard library routines are covered in Chapter 19. Each routine lists the interface file you must include in your program to use the function.) 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 - CC should be checked. If it is not, select CC 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.)

```
/*************************************************
*
*  Finance
*
*  This program prints the balance on an
*  account for monthly payments, along with the
*  total amount paid so far.
*
*************************************************/

#include <stdio.h>
```

```
#define LOANAMOUNT 10000.0       /* amount of the loan */
#define PAYMENT    600.0         /* monthly payment */
#define INTEREST   15            /* yearly interest (as %) */

int main(void)

{
float balance,                   /* amount left to pay */
      monthlyInterest,           /* multiplier for interest */
      paid ;                     /* total amount paid */
int   month;                     /* month number */

/* set up the initial values */
balance = LOANAMOUNT;
paid = month = 0;
monthlyInterest = 1.0 + INTEREST/1200.0;

/* write out the conditions */
printf("Payment schedule for a loan of %10.2f\n", LOANAMOUNT);
printf("with monthly payments of %5.2f at an\n", PAYMENT);
printf("interest rate of %d%%.\n\n", INTEREST);
printf("          month        balance    amount paid\n");
printf("          -----        -------    -----------\n");

/* check for payments that are too small */
if (balance*monthlyInterest - balance >= PAYMENT)
   printf("The payment is too small!");
else
   while (balance > 0) {
      /* add in the interest */
      balance *= monthlyInterest;
      /* make a payment */
      if (balance > PAYMENT) {
         balance -= PAYMENT;
         paid += PAYMENT;
         }
      else {
         paid += balance;
         balance = 0;
         }
      /* update the month number */
      ++month;
      /* write the new statistics */
      printf("%15d %14.2f %14.2f\n", month, balance, paid);
      }
}
```

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 .CC to the name. For this particular program, save it as Finance.cc.

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.
*
*************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <misctool.h>

#define FALSE 0                 /* boolean constants */
#define TRUE  1

#define BLASTRADIUS 50.0     /* max distance from target for a hit */
#define DTR         0.01745329  /* convert from degrees to radians */
#define VELOCITY    434.6       /* muzzle velocity */
```

```
int main (void)

{
float angle,                    /* angle */
      distance,                 /* distance to the target */
      flightTime,               /* time of flight */
      x,                        /* distance to impact */
      vx,vy;                    /* x, y velocities */
int   done,                     /* is there a hit, yet? */
      tries,                    /* number of shots */
      i;                        /* loop variable */

/* choose a distance to the target */
srand((int) time(NULL));
for (i = 0; i < 100; ++i)
   rand();
distance = rand()/5.55373;

/* not done yet... */
done = FALSE;
tries = 1;

/* shoot 'til we hit it */
do {
   /* get the firing angle */
   printf("Firing angle: ");
   scanf("%f", &angle);

   /* compute the muzzle velocity in x, y */
   angle *= DTR;
   vx = cos(angle)*VELOCITY;
   vy = sin(angle)*VELOCITY;

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

   /* find the distance */
   /* (distance = velocity*flightTime) */
   x = vx*flightTime;

   /* see what happened... */
   if (fabs(distance-x) < BLASTRADIUS) {
      done = TRUE;
      printf("A hit, after %d", tries);
      if (tries == 1)
        printf(" try!\n");
      else
        printf(" tries!\n");
      switch (tries) {
         case 1:
            printf("(A lucky shot...)\n");
            break;
         case 2:
            printf("Phenomenal shooting!\n");
            break;
         case 3:
            printf("Good shooting.\n");
            break;
         otherwise:
            printf("Practice makes perfect - try again.\n");
         }
      }
   else if (distance > x)
      printf("You were short by %d feet.\n", (int)(distance-x));
   else
      printf("You were over by %d feet.\n", (int)(x-distance));
   ++tries;
   }
while (!done);
}
```

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.

Library Functions

Your most important resource for writing programs that run in the text environment are the library functions. These are available in any program you write – to use them, you simply include the appropriate header file. Library functions include disk access routines like `fopen`, `fscanf`, and `fprintf`; mathematical functions like `cos`, `atan`, and `tan`; and many others, like `malloc` and `free`. These functions are described in Chapter 19.

If you have used many C compilers, you know that the libraries vary from one compiler to another. This is because the C libraries have evolved somewhat with time, and because the complete set of libraries is so large that some implementors don't bother with a few of the obscure functions. Our libraries are drawn from four sources: the ANSI C standard, APW C, and a few we added that are unique to ORCA/C. With a few minor exceptions, all of the ANSI C libraries are implemented in ORCA/C. APW C is an older C compiler written by Apple Computer. It included a few extensions to the standard C libraries to deal with the Apple IIGS. To make it easier for people with APW C to move their programs to ORCA/C, we have implemented these extensions to the standard C libraries exactly as APW C did. There are also a few libraries that we added to ORCA/C to make it easier to use the toolbox.

For a complete description of the ORCA/C libraries, as well as examples on how to use them, see Chapter 19.

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 consol control codes are covered in Appendix D.

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.

Programming in Pieces

When you are writing small, self-contained programs, the program is generally written in a single source file, and compiled as a single piece. There are two occasions when you might want to break a program up into more than one piece.

Most programmers find that they write the same sorts of programs over and over. Some people like to write games, others use their computers for engineering calculations, and some write programs to work on data from their business. In any case, as you start to write more and more programs, you may find that you are using a few subroutines in several programs. You can, of course, cut the subroutines from one program file and paste it into the new one. There is another alternative, though, that you should consider. You can create a library of your commonly used subroutines, and compile it once. From that point on, to use your subroutines in a new program, you simply need to place a `#include` directive at the top of your program. This cuts down on the amount of disk space you need for your program, since each program's source file is shorter. It also cuts down on compile times, since the compiler doesn't need to recompile the subroutine each time you change your program.

Splitting a program up into a number of small sections, each with subroutines that perform similar functions, is a popular way to organize a large program to make it easier to understand. This cuts down on the size of each individual file the compiler needs to compile, reducing the time required to recompile a program. It also reduces the amount of memory needed to compile a large program.

To get a firm grasp on separate compilation, we will create a library of trigonometric functions, something that would be useful for an engineering student. We will implement three trig functions: cosecant, secant, and cotangent. These functions are the inverses of the common trig functions sine, cosine and tangent. The three functions will appear in a separately compiled module.

A separately compiled module has many similarities to a program, but there are some important differences, too. The most important is that a separately compiled module does not have a function called `main`. In any C program, `main` is the first function called. A separately compiled module is called from another module or from a C program, so it should not have a function by that name.

When you write a module, you need to write two separate source files. The first part is the header. It is used by other modules and programs, and tells them what functions and variables exist within the module. By convention, we generally name the header file with a .h at the end of the file name. In our trig example, we will name the interface file Trig.h. The header file, shown below, has one definition for each of the three functions we will create. Also, note how the __TRIG__ macro is defined: it causes the definitions to be skipped if they have already been defined. That helps in cases where you may use your module both from another interface file and from the main program. By automatically skipping the definitions if they have already been defined, you avoid duplicate definition errors.

```
#ifndef __TRIG__
#define __TRIG__

extern double secant(double r);
/* compute the secant of the argument */
extern double cosecant(double r);
/* compute the cosecant of the argument */
extern double cotangent(double r);
/* compute the cotangent of the argument */

#endif
```

If you have never used ANSI C, it may seem a bit odd to see the type of the parameter in the argument list. This is a new feature in ANSI C called function prototypes. It allows the compiler to check to make sure that when you call one of these functions you pass the correct number and type of parameters. Using function prototypes in all of your interface files can cut your debugging time enormously, since the compiler can catch errors that cause crashes in programs compiled with older compilers. We highly recommend their use, but ANSI C does not require the use of function prototypes.

The second part of a separately compiled module is the actual executable code and variables. It looks just like a program, but it doesn't have a function called `main`. By careful use of the storage class static (explained fully in Chapter 15), you can create functions and variables that can only be used from within the module, and that won't conflict with anything from another module by the same name. While it is not strictly required by C, we recommend that you always hide any function or global variable that does not appear in the interface file.

The example below shows the implementation part of the trig module. Go ahead and type it in, and save it using the file name Trig.cc. While you can choose your own names, just as with a program, the examples in the rest of this section assume you have used the file names Trig.cc and Trig.h.

```
#include <math.h>

double secant(double r)
/*compute the secant of the argument*/

{
return 1.0/cos(r);
}

double cosecant(double r)
/*compute the cosecant of the argument*/

{
return 1.0/sin(r);
}

double cotangent(double r)
/*compute the cotangent of the argument*/

{
return 1.0/tan(r);
}
```

Compiling a module is a little trickier than compiling a program. First, modules must be compiled to disk – you will need to link the object module into you program, so you must ask the compiler to save the object module to disk. Second, modules are not programs, so you cannot link or execute them.

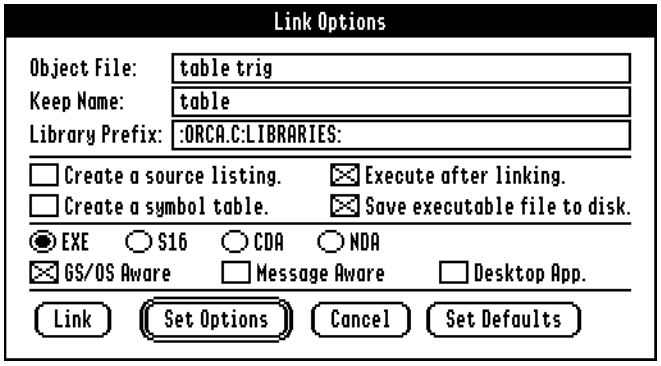
The first step in compiling the module is to pull down the Run menu and select the Compile command. When the compile dialog appears, turn off the Link after compiling option by clicking on the box to the left of the option name. There shouldn't be an X in the box when you're done. You can leave the debug option on if you like, but if you are creating a library, you should probably turn debugging off after the library has been debugged. Now click on the Set Options button. To compile the unit, use the Compile to Disk command from the Run menu – Compile to Memory should not be used with modules.

The last step is to delete the .root file created by the compiler. If you forget this step, the program will still work, but it will be a bit larger than it needs to be. In our example, the .root file is called Trig.root. Use the Delete command from the Finder or the ORCA shell to delete this file. You could also use the noroot pragma, described in Chapter 12, to avoid creating a root file in the first place.

Eventually, we will make a library from this module, but for now, let's treat the module as a part of a separately compiled program, just to see how that is done. Start by opening a new window and typing in the program shown below. Be sure to check to make sure the language is CC.

```
#include <math.h>
#include <stdio.h>
#include "trig.h"

#define pi 3.14159265358979323846

int main (void)

{
double angle;

angle = pi/20.0;
printf("%25s%25s%25s\n", "secant", "cosecant", "cotangent");
while (angle < pi/2.0) {
   printf("%25e%25e%25e\n", secant(angle), cosecant(angle),
      cotangent(angle));
   angle += pi/20.0;
   }
}
```

Save the file as Table.cc. Go ahead and compile this program, too. As before, you should use the Compile to Disk command, and you should not link after compiling. Once compiled, you will need to link the two pieces together. To do this, pull down the Run menu and use the Link command. Type the names of the two files into the Object File line; the names used in our example would be "table trig". It is important to place the name of the program first, and to separate the names with a space. In addition, you must place the name of the output file – TABLE in this example – in the Keep Name line. The completed dialog looks like the one below. Once the file names have been typed in, click on the Link button to link the programs together. Once the program has been linked, it will execute automatically.


Link Options

Object File: table trig
Keep Name: table
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.

As you can see, separate compilation isn't nearly as easy as writing a program in one piece. There is an easier way to handle separate compilation, but it requires learning to use the shell. The shell is covered in Chapters 6 and 8. It's also easier to use a library from a program, although there is still a bit of work involved in setting the library up. To turn TRIG into a library, we will start by using the MAKELIB utility. The MAKELIB utility is used to create a library from one or more object files. It could, for example, combine two units into a single library. In this section, we will only cover creating a new library. See the description of the MAKELIB utility in Chapter 8 for details on how to use the other capabilities of the utility.

So far, we haven't used any utilities, so let's cover that first. Utilities are text based programs that are executed by typing their name in the shell window. To use the MAKELIB utility, start by selecting the shell window. Now type

```
makelib triglib +trig.a
```

and press the return key. Be sure to type the line with spaces between the three names, but don't put a space between the + character and TRIG.A. You can use uppercase or lowercase letters. This will create a library file called TRIGLIB on your disk.

The next step is to install this library in the library prefix. The order that libraries appear in the library prefix is important, so the way you copy your new library into the library prefix is also important. Basically, your library should come before the libraries already in the library prefix. To accomplish this, you need to copy all of the library files in the library prefix to another prefix, then delete them from the library prefix. Next, you will copy your library into the library prefix. Finally, you will copy all of the old libraries back into the library prefix, and delete them from the work prefix. You do not have to copy the ORCACDefs folder – its order in relation to the libraries is unimportant. Once you have done this, the linker will automatically scan the TRIGLIB file when it links your programs, pulling any subroutines needed from the library. In addition, you need to copy the file TRIG.H into the ORCACDefs folder. With the TRIG.H file in the ORCACDefs folder, you can use the <> characters around the file name, as in

```
    #include <trig.h>
```

When you use the <> characters, the compiler always looks in the ORCACDefs folder for the interface file, allowing you to use the interface file no matter which directory or disk your source program is on. If you use the "" characters around the file name, as we did in the example above, the compiler looks in the current directory for the interface file.

If you are familiar with text shells, you can do all of this copying from the shell window itself. If not, leave the ORCA environment and move back to the Finder, doing the copying from the Finder.

If you run out of disk space during these operations, move the samples folder from the source disk to your program disk, and delete it from the ORCA/C program disk.

While the steps involved in creating the library are involved, using a library is very easy. The sample program you entered a few minutes ago needs one small change: replace the "" characters around the file name trig.h with <> characters. Now you can turn Link after compile back on in the Compile dialog, and use the Compile to Memory command again. You can also use the trig library from any other program you write in the future, without recompiling the trig library.

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 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/C 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/C, but there is a lot more to QuickDraw II than you see here.

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

```
#include <quickdraw.h>
```

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.

```
#include <quickdraw.h>
#include <math.h>

int main (void)

{
float r, theta;

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

Save the program as Spiral.cc. 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/C automatically initializes several tools, and opens the .CONSOLE device used for text input and output. Graphics programs, though, are using QuickDraw II, and ORCA/C 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 library function startgraph. The startgraph function 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 library function endgraph to shut down the graphics environment. Finally, you must include orca.h in your program. This is the interface file that we use for the library functions we have added to ORCA/C that are not a part of the tools or of ANSI C. The spiral program is shown below, changed to meed these requirements. The changes are shown in bold-face.

```
#include <quickdraw.h>
#include <math.h>
#include <orca.h>

int main (void)

{
float r, theta;

startgraph(640);
theta = 0.0;
r = 40.0;
SetPenSize(2, 1);
MoveTo(280, 40);
while (r > 0.0) {
   theta += 3.1415926535/20.0;
   LineTo((int) (cos(theta)*r*3) + 160, (int) (sin(theta)*r) + 40);
   r -= 0.15;
   }
endgraph();
}
```

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 C. If you don't already know how to program in C, *learn C 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 C, but you should feel comfortable writing programs that are a few hundred lines long, and you should understand how to use structures 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 C 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 header files, too. While ORCA/C 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 header 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 Header 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 .h header file 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 tool header file will include other tool header files to get the additional declarations.

There are actually two different versions of the tool header files. One version, which you can find on the More Extras disk at the path :MoreExtras:Apple:Libraries:APWCInclude, was created by Apple Computer for the APW C compiler, which is an older, pre-ANSI C compiler. We've included these headers in case you need them for compatibility with some older programs. The headers you will normally use with ORCA/C will be installed for you when you install ORCA/C on a hard disk. These headers are derived from Apple's but they are prototyped, so ORCA/C can check to make sure you are passing the correct number and kind of parameters.

Here's a list of the current toolbox header files and the tool they define. A few, like GSOS.H and FINDER.H, don't technically document the tools, but they are included here for completeness.

Interface File Name	Tool
ACE.H	Audio Compression/Expansion
ADB.H	Apple Desktop Bus
APPLESHARE.H	AppleShare Tools
APPLETALK.H	AppleTalk Tools
CONTROL.H	Control Manager
DESK.H	Desk Manager
DIALOG.H	Dialog Manager
EVENT.H	Event Manager
FINDER.H	Finder Interface
FONT.H	Font Manager
GSBUG.H	GSBug Interface
GSOS.H	GS/OS Disk File Manager
HYPERSTUDIO.H	HyperStudio Interface
HYPERXCMD.H	HyperCard Interface
INTMATH.H	Integer Math Tool Set
LINEEDIT.H	Line Edit Tool Set
LIST.H	List Manager
LOADER.H	Program Loader
LOCATOR.H	Tool Locater
MEDIACONTROL.H	Media Control Tool Set
MEMORY.H	Memory Manager
MENU.H	Menu Manager
MIDI.H	MIDI Sound Tools
MIDISYNTH.H	MIDISynth Tool Set
MISCTOOL.H	Miscellaneous Tool Set
NOTESEQ.H	Note Sequencer
NOTESYN.H	Note Synthesizer

PRINT.H	Print Manager
PRODOS.H	ProDOS Disk File Manager
QDAUX.H	QuickDraw Auxiliary
QUICKDRAW.H	QuickDraw II
RESOURCES.H	Resource Manager
SANE.H	SANE Floating-Point Tools
SCHEDULER.H	Scheduler
SCRAP.H	Scrap Manager
SHELL.H	ORCA Shell Interface
SOUND.H	Sound Manager
STDFILE.H	Standard File Operations Tool Set
TEXTEDIT.H	Text Edit Tool Set
TEXTTOOL.H	Text Tool Set
TYPES.H	types and constants
VIDEO.H	Video Tools
WINDOW.H	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 high level language debugger and your program need the mouse, keyboard, and menu bar to function.

As soon as the high level language 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 4.2 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. 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
- Quick Draw Aux
- 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 functions 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:

```
#pragma nda 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 function 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. The function does not return a value.
action	The action parameter is the name of a function that is called whenever the desk accessory must perform some action. It must declare a long parameter whose type varies based on the action type, followed by an integer parameter which defines the action that the function should take. 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. The function returns a boolean indicating if the action was handled.
init	The init parameter is the name of a function 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 function must define a single integer parameter. The function will be zero for a shut down call, and non-zero for a start up call. The function does not return a value.
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**.

In other C programs, the program always starts executing at the function main. This is not true in desk accessories. So, unlike other C programs, desk accessories do not have to have a function called main.

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

```
#pragma nda StartUp ShutDown Action Init 60 1023 "  Sample\\H**"

#include <window.h>

GrafPortPtr StartUp(void)

{
<<<open the window and assign the pointer>>>
}


void ShutDown(void)

{
<<<close the window>>>
}


int Action (long param, int code)

{
<<<handle events>>>
}
```

```
void Init(int code)

{
if (code)
   <<<startup code>>>;
else
   <<<shutdown code>>>;
}
```

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 to turn debugging off for your final compile!

For a sample desk accessory that illustrates these principles, see the CLOCK program in the DESKTOP.SAMPLES folder of 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 C'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 `#pragma cda` 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 name of the function that is called when your program is selected from the list of available classic desk accessories. The start function serves the same purpose as the function main in a normal C program.
shut down	Shut down is a function returning void 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.

```
#pragma cda "Echo from C" 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 1989
*  Byte Works, Inc.
*
*****************************************************************/

#include <stdio.h>

char str[256];

void Start(void)

{
printf("This program echoes the strings you "
       "type from the keyboard.  To\n");
printf("quit, hit the RETURN key at the beginning of a line.\n\n");

do {
   fgets(str, 256, stdin);             /* read a string */
   printf("%s\n", str);                /* write the same string */
```

```
      }
   while (strlen(str) > 1);            /* quit if the string is empty */
   }


   void ShutDown(void)

   {
   }
```

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 functions that will be called; you can use any legal C 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, comment out the `#pragma cda` directive, and then create a main function with two statements, one to call the Start function, and a second to call the ShutDown function. The function main for the echo sample, shown above, would be

```
   main()

   {
   Start();
   ShutDown();
   }
```

After the program is debugged, remove main and the CDA 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/C.

When most C programs are complete, ORCA/C 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 pragma at the start of the program. The rtl pragma has no parameters. As with other C programs, your program starts executing at the function `main`.

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 C 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/C-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/C with the aid of the nba pragma and the HyperStudio.h header file. The format for the nba pragma is:

```
#pragma nba main
```

main — This parameter is an identifier that specifies the name of the function that is called when HyperStudio calls the NBA. This function accepts one parameter of type HSParamPtr. The function returns void.

The parameter that is passed to the function is a pointer to the HyperStudio parameters, contained in a struct. The structure itself is defined in HyperStudio.h.

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 function, 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 HyperStudio.h header 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/C with the aid of the xcmd pragma and the HyperXCMD.h header file. The format for the xcmd pragma is:

```
#pragma xcmd main
```

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

The parameter that is passed to the function is a pointer to the HyperCard parameters, contained in a struct. The structure itself is defined in HyperXCMD.h.

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.h 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/C with the aid of the cdev pragma. The format for the cdev pragma is:

```
#pragma 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 two long integer parameters and an integer parameter, 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

There are two ways to mix assembly language with ORCA/C. One way is to use the built-in mini-assembler that is a part of ORCA/C. The mini-assembler is suitable for tasks where a few lines of assembly are needed in a program that is mostly C. When several dozen lines or several subroutines must be coded in assembly language, you can also choose either the ORCA/M macro assembler, which offer more power larger assembly language tasks. This chapter deals with using the ORCA/M macro assembler to write entire functions or libraries in assembly language. Chapter 18 gives details on using the asm statement to embed assembly language code in a C program.

To understand all of the information in this chapter, you must already know assembly language. You will also need to know how to use the assembler itself, and you must install the assembler in your C development system.

The Basics

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

We will define a small function to clear the keyboard strobe. This is one of those tasks that is difficult to do from C, yet takes only four lines of assembly language. You might want to call this function 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 scan library call.

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

```
extern void Clear(void);

int main(void)
{
Clear();
}

#append "myprog.asm"
```

Once you have typed the program in, save it as `MYPROG.CC`. Be sure the language stamp is CC. 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 C 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.

It is also possible to separately assembly the assembly language file and use the linker to compile the assembly language and C portions of the program, much as we did in the last chapter, when the trig example was used to show how a program made up of two or more C source files could be linked into a single executable program. From the desktop development environment, it is easier to use the method we are describing here. If you prefer to use separate compilation, use the same methods discussed in the last chapter.

At this point we need to add the assembly language function. 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 function shown below.

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

An important point to remember is that C is one of the few case sensitive languages ever written. Like most languages, assembly language is normally case insensitive. The case on directive at the start of the assembly language program makes the assembler case sensitive, like C. If you leave this directive out, you must always use uppercase characters when referring to the function from C; you must also use uppercase letters when referring to a function written in another high-level language, like Pascal or BASIC.

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

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

1. ORCA looks at the file MYPROG.CC. Since it is a C file, the C compiler is called to compile the program.
2. When the compiler gets to the append directive, it looks at the file named in the operand. Since it is not a C 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 C and assembly language parts into one program and writes an executable file called MYPROG to the current directory.
5. The program is executed.

It is easy to gloss over one detail in this process and end up making a mistake. When you are compiling a program that is made up of more than one language, you must compile to disk, not to memory. It is very important that you use the Compile to Disk command to compile and execute this program, and not the Compile to Memory command. If you use one of the debugger commands (e.g. Step) before a program has been compiled, or after making a change, the program is compiled before the debugger takes over – and it is compiled with a Compile to Memory command. For this reason, it is also important for you to remember to compile your multi-lingual programs before you use one of the debugger commands.

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 C program. Char, int and unsigned int 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, both single and double precision, 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. Structures, unions and arrays are returned as a pointer to the first byte of the object. 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 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 Chapters 13 and 15. 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 C. The C 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.

```
extern void Clear(void);

extern int Keypress(void);

int main(void)

{
while (! Keypress()) ;
Clear();
}
```

```
#append "myprog.asm"
```

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

```
    case   on
;
;  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
```

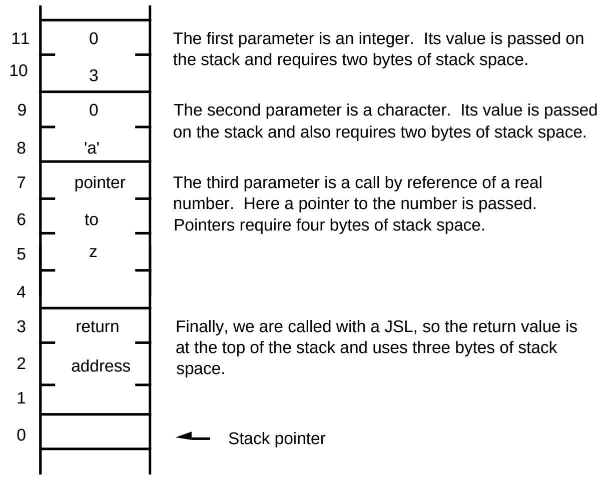
Passing Parameters to Assembly Language Subroutines

To better understand the interaction between C and assembly language in the ORCA environment, we will look at how parameters are passed from a C program to an assembly language subroutine. ORCA/C places the parameters which appear in a subroutine call on the stack, in the opposite of 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 on the type of the value being passed. Int, unsigned int, long, unsigned long, characters, enumerations, structures, unions and pointers appear on the stack as actual values. The normal unary conversions are performed before the value is placed on the stack. Basically, that means that character values are expanded to two bytes before they are pushed, while two-byte and four-byte values are pushed on the stack as is. Structures and unions are placed on the stack in the same format that they are stored in memory, while floating-point values are pushed on the stack as extended format SANE numbers. Arrays and string parameters (which are actually just a special form of an array) are passed as an address that points to the first byte of the value.

Consider the C program fragment below:

```
...

void doSomething(z, ch, i);

int i;
char ch;
float *z;

...

i = 3;
ch = 'a';
*z = 5.6;
doSomething (&z, ch, i);

...

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 C 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/C to alleviate the burden of manipulating the stack frame from assembly language.

```
/* Demonstrate calling assembly language functions from C. */

extern int reverse(int parm);

int main(void)

{
printf("%d %d\n", 6, reverse(6));
}

#append "reverse.asm"
```

```
         case  on
;
;  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 CSUBROUTINE and CRETURN, and are located in a file called M16.CC. You can find this file on the extras disk in the Libraries:AInclude folder.

CSUBROUTINE 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 C 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 one.

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

```
void Sample (int *i, float r, char c);
```

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 subroutine:

```
Sample    start
ptr equ   1       work pointer

     csubroutine (4:i,4:r,2:c),4
```

Notice that i is a pointer to an integer, so you pass a pointer, not an integer. Pointers are four bytes long. The parameter r is described to the CSUBROUTINE 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 or double number as a parameter, however, we pass the ten-byte value itself on the stack. Finally, character values require two bytes.

The CRETURN macro has a similar protocol. If you are writing a function returning void, 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 your direct page area or in a variable that can be accessed with absolute addressing. For example, to return a pointer called PTR, code

```
     creturn 4:ptr
```

If a function is returning a floating-point value, return a pointer to a SANE extended format number. If the function returns a structure or union, return a pointer to the first byte of the structure or union.

Accessing C Variables from Assembly Language

All global variables which can be accessed from outside of a C source file by another C function are available from assembly language. In addition, if an assembly language file is appended to the end of a C source file using the append directive, all global variables within the C source file, even if they are private to the file, can be accessed from the assembly language subroutines appended to the file. If you are using the small memory model, all C variables are accessed using absolute addressing. In the large memory model all variables must be accessed using long addressing.

It is also possible to define variables in assembly language that can be accessed from C. To do that, define the variable as external in the C program, just as you would if the variable was in a separately compiled C function. 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.

The C language uses case sensitive variable names. The assembler defaults to case insensitive variables, passing all variable names to the linker as uppercase strings. To make the assembler case sensitive, use the assembler's case directive.

For details on how the variables are stored in memory, see Chapters 13 and 15.

Calling C Procedures and Functions from Assembly Language

Calling a C function from assembly language is extremely straightforward. You simply push any required parameters onto the stack and issue a JSL to the function you want to call. The one trick is that you must push the parameters on the stack staring with the last parameter, and working toward the first. 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 and double values are returned as pointers to ten-byte SANE extended format numbers. Note that real numbers should be passed as ten-byte SANE extended format numbers. Structures, arrays and unions are also returned as a pointer to the first byte of the object.

For example, to call the C function

```
int cfunc(myStruct *ptr, int i);
```

that takes a pointer and integer as input and returns an integer result, you could use the ph2 macro (supplied with ORCA/M) to push the integer variable, the ph4 macro to push the pointer onto the stack, and then call the function, as follows:

```
         ph2    i      push the integer
         ph4    #parm  push the address of the structure
         jsl    cfunc  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. There are several other advantages, at least with the current software. The text environment does not support the source level debugger. Its editor can only show you one file at a time, and each file is limited to 64K bytes. In addition, you cannot split the screen, looking at two parts of the file at the same time.

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 C 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 C with assembly language, 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/C, something you can do with an installer script; see Appendix B for details if you are interested in doing this.

The other thing you can do is to set up ORCA/C 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/C 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/C, 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 output." 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

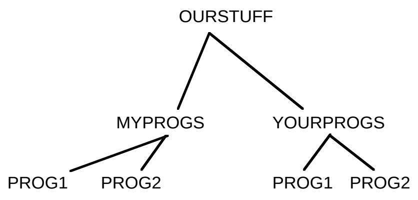
```
prefix :mydisk
prefix myfolder
```

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.C:
@   :ORCA.C:
8   :ORCA.C:
9   :ORCA.C:
10  .CONSOLE:
11  .CONSOLE:
12  .CONSOLE:
13  :ORCA.C:LIBRARIES:
14  :ORCA.C:
```

```
15 :ORCA.C:SYSTEM:
16 :ORCA.C:LANGUAGES:
17 :ORCA.C:UTILITIES:
18 :ORCA.C:
```

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.C. You could use the path name

```
18:myprog
```

and it would have the same effect as

```
:orca.c: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/C.

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 C compiler looks in this folder for another folder called ORCACDefs to resolve include 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.C directory. To do this, type

```
disable d =
```

Cataloging :ORCA.C 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.C 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/C 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/C that can be installed in your system folder using the Installer. Once installed, your C programs can redirect output to .PRINTER to print files, or even open .PRINTER as a file from within a C 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/C userid function. This library 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 library function shellid, which returns a pointer to a null-terminated string. If the program launcher does not provide a shell identifier, the function returns a NULL pointer. 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 library function commandline, which returns a pointer to a null-terminated string containing the character in the command line. If the program launcher did not provide a command line, the commandline function returns NULL.

The C language provides another mechanism for reading the command line, via the arguments argv and argc to the main function. You can use this feature in your program. The principal difference between using argc and argv verses using the commandline function is that argc and argv break the command line up into words, while the commandline function gives you the command line exactly as it was typed.

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 :CC.Samples:Text.Samples:CLine.cc. 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.C: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 cancelled 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

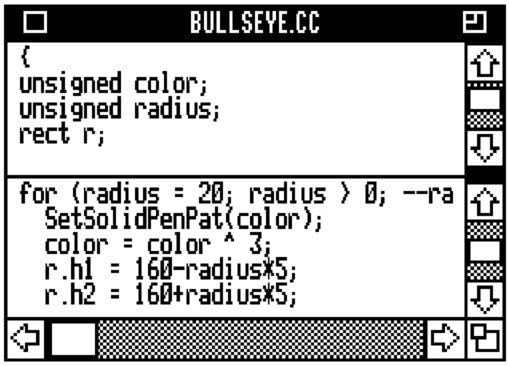
Word selection allows you to quickly isolate a single word. To do this, move the cursor so that it is on the word you want to select, and click the mouse rapidly two times. This is called double-clicking. The word that the mouse was on is selected.

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



BULLSEYE.CC

```
{
unsigned color;
unsigned radius;
rect r;
```

```
for (radius = 20; radius > 0; --ra
  SetSolidPenPat(color);
  color = color ^ 3;
  r.h1 = 160-radius*5;
  r.h2 = 160+radius*5;
```

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.

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

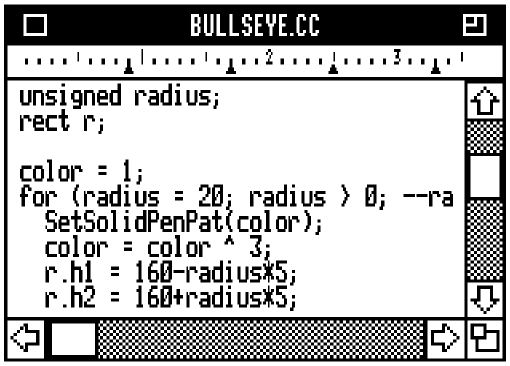
You can move to the start of a line by holding down the key and typing the LEFT-ARROW key. This moves to the first column in the line, regardless of the current auto-indent mode. To move to the end of the line, hold down the key and type the RIGHT-ARROW key. This moves to the column immediately after 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.



```
unsigned radius;
rect r;

color = 1;
for (radius = 20; radius > 0; --ra
  SetSolidPenPat(color);
  color = color ^ 3;
  r.h1 = 160-radius*5;
  r.h2 = 160+radius*5;
```

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.

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.

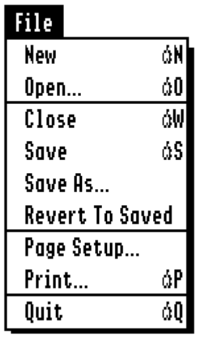
Default tab stops

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 C. 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.

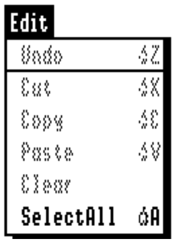
The Print command brings up a standard dialog to control the printing process. This dialog is documented in the manuals that came with your computer.

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/C.

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 at 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.

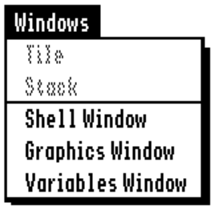
Select All

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

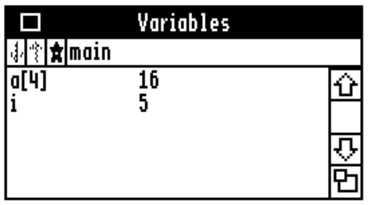
The Shell Window command opens the shell window. The shell window is basically an untitled window with a few special characteristics. The special characteristics are: the shell window has the name Shell, rather than a name that starts with Untitled; it shows up in the top right corner of the screen; 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 are not selectable 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 function main, 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 function main. If you select the up-arrow, you will see the variables display that you created for main, and the down-arrow will now be selectable 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 structure or union by typing the structure or union name, a dot, and the name of the field.

If a pointer points to a structure or union, you can look at a field in the structure or union 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 structure 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 (C supports one byte integers, but displays them as characters. Enumerations are displayed as 2-byte integers.)
- 4-, 8-, and 10-byte reals
- Pascal style strings and null terminated strings
- booleans (Boolean variables are not used by C. In Pascal, their value will be printed as either true or false. The debugger only looks at the first byte of a boolean – a zero is regarded as false, while any other quantity is regarded as true.)
- characters (As with booleans, 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.

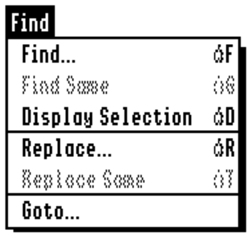
If you would like to use a different window, you can select it from the windows list. The window you select is placed on top of all of the other windows on the desktop, and becomes the active window.

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

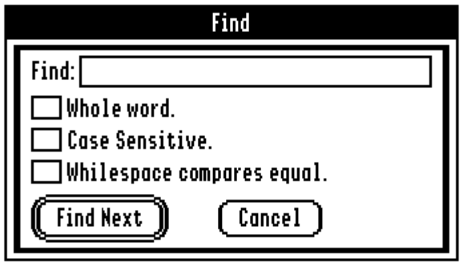
The Find menu helps you locate strings in a window, replace occurrences of a string with another string, find the cursor, or move to a particular line by line number.

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

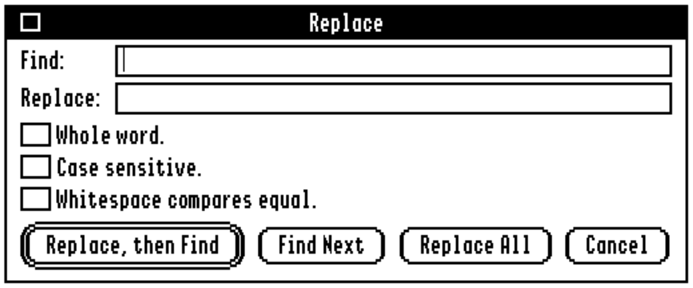
If you have already entered a search string using the Find command, you can search for the next occurrence of the same string using the Find Same command. This allows you to avoid using the Find dialog, and enables searching by simply using the G keyboard equivalent.

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

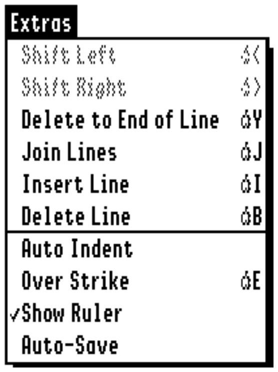
The Goto command lets you move to any line in the open file by specifying a line number. The line number is entered as a decimal value in the Goto window's line-edit box. Clicking on the Goto button causes the desired line to appear at the top of the window, with the insertion point changed to the beginning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

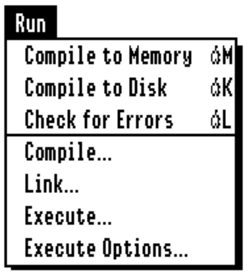
The Auto Save option is a safety measure. If you execute a program, and the program crashes, you cannot return to the desktop to save your files. Any changes that have been made to the files since the last time they were saved to the disk are lost.

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 C's malloc and free functions. 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 disks 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 C 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/C 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

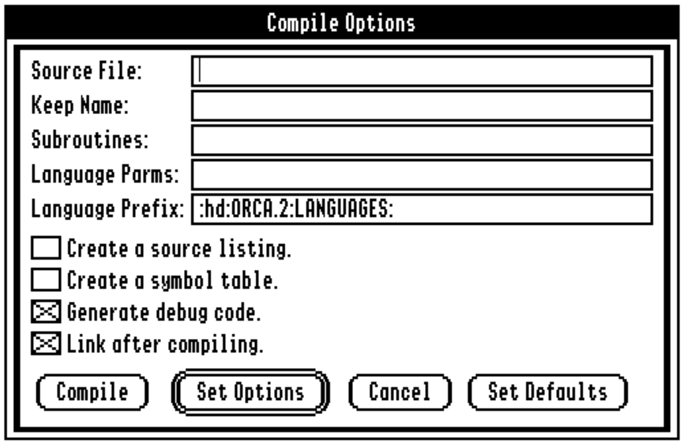
The Check For Errors command compiles your program, but does not save the result of the compile. This allows the compiler to scan your program quickly for errors. Most compilers can scan for errors about twice as fast as they can compile a program. Once all errors have been removed, you can use one of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


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

- [ ] Create a source listing.
- [ ] Create a symbol table.
- [x] Generate debug code.
- [x] 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/C 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.

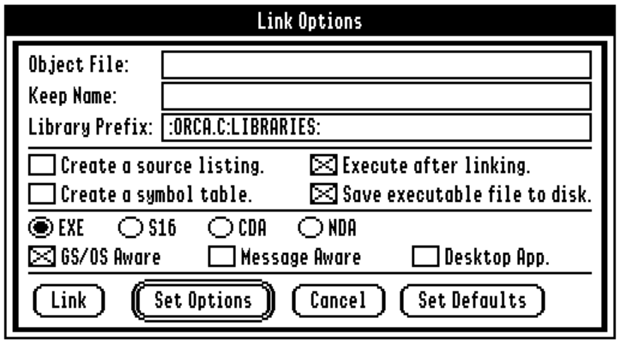
Checking the Create a source listing box causes the compiler to produce a listing of your source file as it compiles your program, and checking the Create a symbol table box causes the compiler to produce a symbol table. A symbol table is a summary of the all of the functions and variables detected in the program. ORCA/C 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.CC, 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 libraries 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 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.

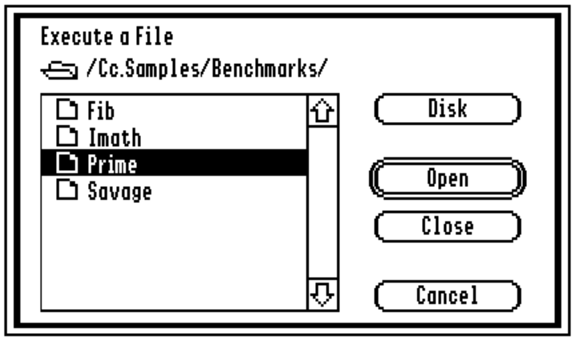
Classic desk accessories cannot be debugged directly from the desktop. To debug a classic desk accessory, compile it as an EXE program with a main program that calls the initialization and action functions. Once debugged, remove the main program from the source code, turn off debugging, change the file 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/C 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

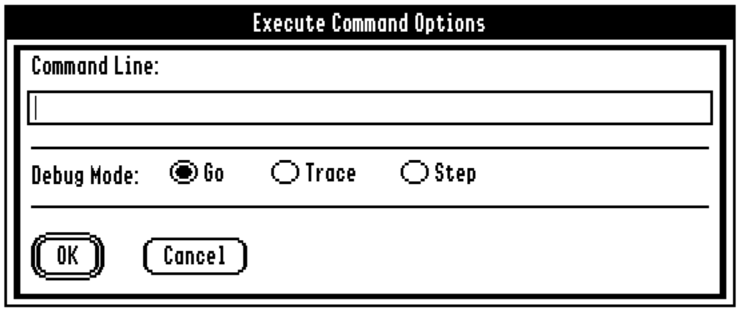
The Execute command allows you to run an executable program. The program's file type must be EXE, S16, or SYS. The dialog box brought up by this command is shown below:



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.


Execute Command Options

Command Line:

Debug Mode: [✓ Go] Trace Step

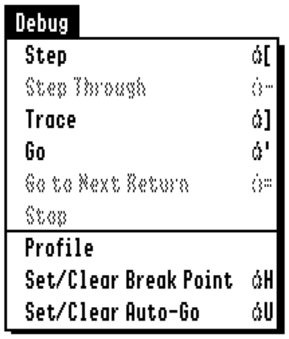
OK Cancel

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.

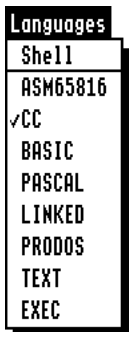
Set/Clear Auto-Go

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 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 file sets 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 character is reserved for future enhancements.

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/C. Note that only the first few characters of the tab line are shown; the tab line actually contains 255 characters.

```
8
100101
0000000010000000010000000010000000010000000010
```

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:

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 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.
ESC X 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 cancelled. 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 are restartable, 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 Pascal 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 C 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.

The C compiler looks for a folder called ORCACDefs in the library prefix when <> characters are used around a file name in a #include statement. The path searched is 13:ORCACDefs.

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/C 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 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. TYPE command;

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:

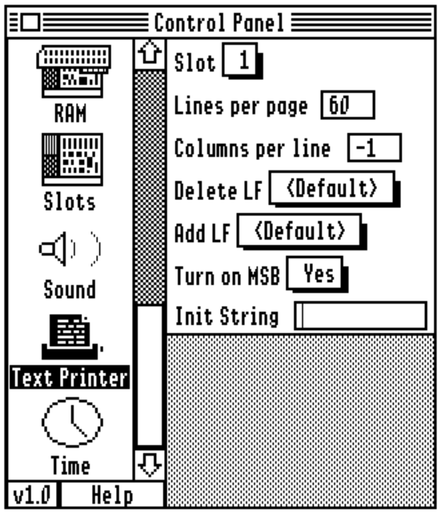
```
TYPE MyFile >.Printer
```

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.

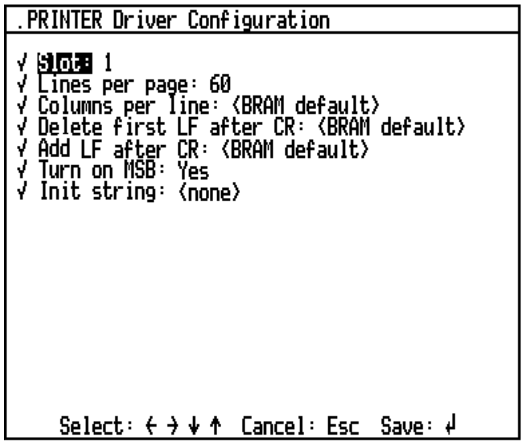
The only thing you have to watch out for is that, since .PRINTER is a RAM based driver, it must be installed on your boot disk before you can use the driver. If you are running from the system disk we sent with ORCA/M, the .PRINTER driver is already installed, and you can use it right away. If you are 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/C 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/M. 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 SYSPIPEn 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 SYSPIPEn 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 is restartable. That is, it need not be reloaded from disk every time it is invoked. The ORCA/C compiler, linker, and editor are all restartable.

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, restartable utilities are denoted in the command table by preceding the U with an asterisk. Restartable programs 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 are restartable! You should not mark a program as restartable unless you are sure that it is, in fact, restartable.

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
SHUTDOWN	C	47	shut down the computer
SWITCH	C	33	switch order of files
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/M 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, ...Dn 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 inclosing 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/C.

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/M assembler, 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 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. See "Precompiled Headers" in Chapter 12 for a complete description of these files and how they are used.
+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	END directive;ORCA/C 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. See "Precompiled Headers" in Chapter 12 for a complete description of these files and how they are used.
+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/C 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 segments to be assembled or compiled. Separate the segment names with one or more spaces. The ORCA Linker automatically selects the latest version of each segment when the program is linked.

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

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/C compiler supports the following options:

-d This option defines a single-token macro. It is followed immediately by the macro name, with no imbedded spaces, an equal sign, and then by a single preprocessor token, or a number with a leading sign. Multiple macros can be defined. For example

```
cc=(-dfoo=-1 -dbar="bar")
```

is completely equivalent to starting the source file with

```
#define foo -1
#define bar "bar"
```

-i This option adds a pathname to the path search sequence. It is followed by a pathname, given as a string. Using this option is completely equivalent to placing a path pragma at the start of the program. For example

```
cc=(-dfoo=4 -i"..:myheaders:")
```

is completely equivalent to starting the source file with

```
#define foo 4
#pragma path "..:myheaders:"
```

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

```
ASSEMPLE     [+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.

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 restartable programs that have 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/M 2.0 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 is 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 redirectable 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 C 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 fully relocatable programs, and they load faster, since all two-byte relocation information can be resolved at link time, rather than creating relocation records for each relocatable 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/C 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.

While you can set the language and create Pascal files, you will not be able to compile them unless you purchase the ORCA/Pascal compiler and install it with ORCA/C.

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 page 96 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|-N] [+T|-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.

*startline*n The line number of the first line of this file to be printed. If this parameter is omitted, then the entire file is printed.

*endline*n 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 my 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:

CTRLkey	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 OPTIONkey 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.

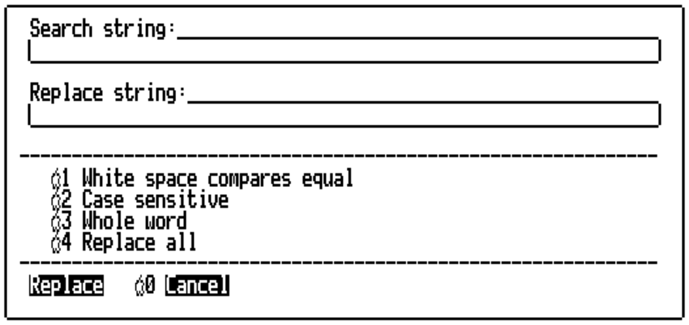
Macros are saved on disk in the file SYSEMAC in the ORCA shell prefix.

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 CTRL Y	**delete to end of line** - Deletes characters from the cursor to the the end of the line.
Z or CTRL Z	**undo** - Resets the string to the starting string.
ESC or CTRL X	**exit** - Stops string entry, leaving the dialog without changing the default string or executing the command.
E or CTRL E	**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.

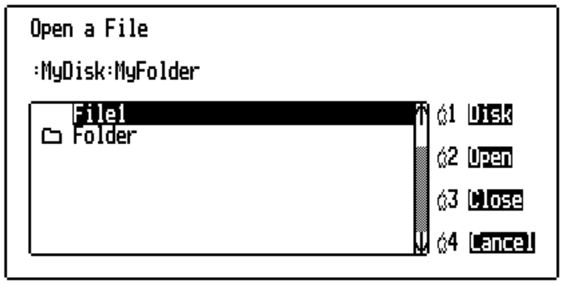


Figure 9.3

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.

CTRLG — 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.

CTRLC 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.

CTRLW 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 CTRL**T**

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

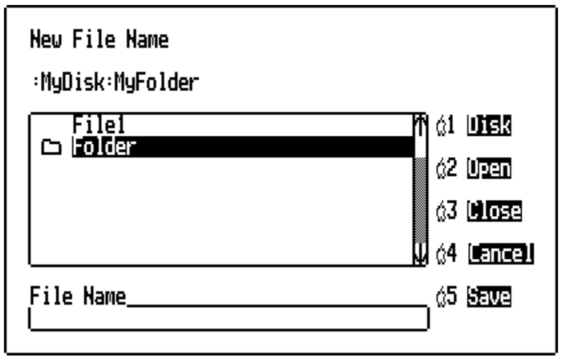
A space is inserted at the cursor position. Characters from the cursor to the end of the line are moved right to make room. Any character in column 255 on the line is lost. The cursor remains in the same position on the screen. Note that the Insert Space command can extend a line past the end-of-line 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.



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.

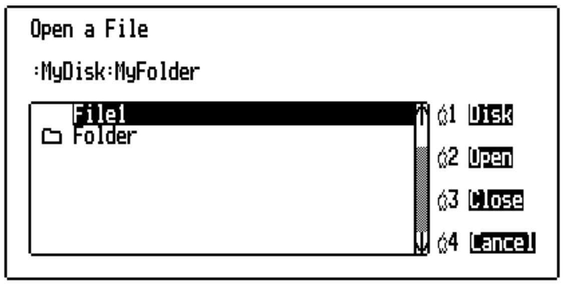


Figure 9.4

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.

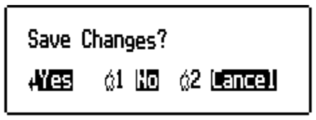
For information on how to use the various controls in the dialog, see "Using Editor Dialogs" in this chapter.

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?

Yes   1 No   2 Cancel

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.

CTRLA 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:

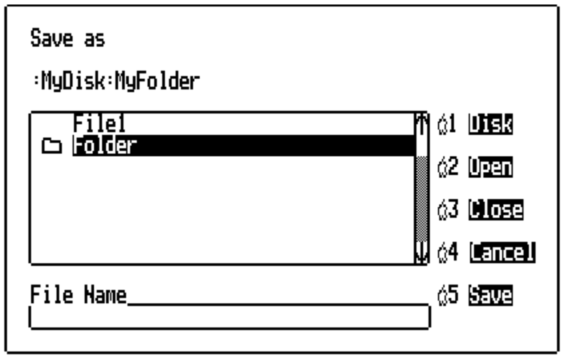


Figure 9.6

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.

CTRL**S** **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

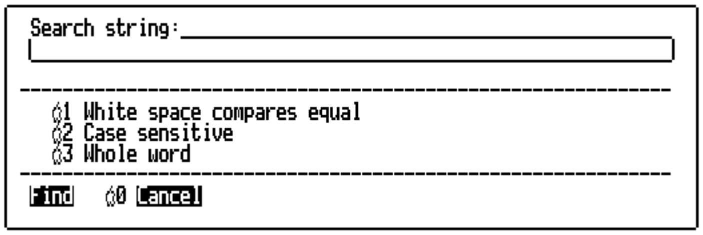
The editor moves up one line in the file, causing all of the lines on the screen to move down one line. The cursor remains in the same position on the screen. If the first line of the file is already displayed on the screen, the command is ignored.

[— 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
------------------------------------------------------------
```

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.

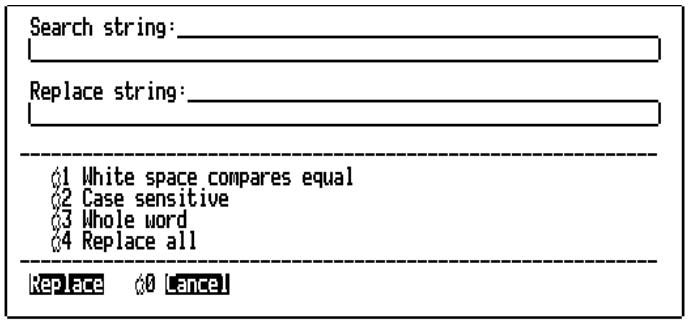
Once a search string has been entered, you may want to search for another occurrence of the same string. ORCA ships with two built-in editor macros that can do this with a single keystroke, without bringing up the dialog. To search forward, use the L macro; to search back, use the K macro.

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:



```
Search string:

Replace string:

--------------------------------------------------------------
   1 White space compares equal
   2 Case sensitive
   3 Whole word
   4 Replace all
--------------------------------------------------------------
Replace   0 Cancel
```

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:

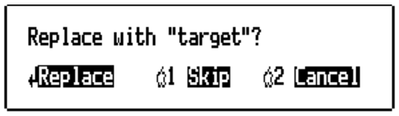

Replace with "target"?

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.

CTRL**Z 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
0000000010000000100000001000000010000000100000001000000000000000100000001000000...
```

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 interspace 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 data 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 {
                                        /* pCount must be at least 6 */
            integer = 3+$$optionalcount (Fields);
            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 columnated 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 0B1111111111111111111111111111111 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.

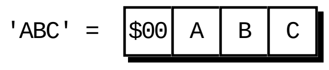


Figure 10.3: Padding of literals

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"
```

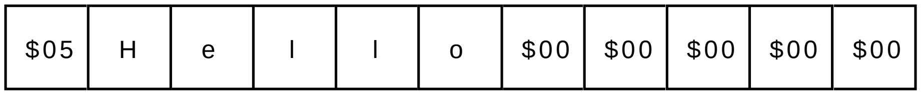


Figure 10.4: Internal Representation of a P-string

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	Hexadecimal	Printable

Sequence	Name	Value	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

C 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

Identifiers in C start with an alphabetic character or underscore and are followed by zero or more alphabetic characters, numeric characters, or underscores. C is a case sensitive language, which means that the identifiers matrix and Matrix are actually two different identifiers.

ORCA/C imposes a limit of 255 characters on the length of any single identifier.

Some examples of legal C identifiers are shown below. They each represent a different identifier.

```
main       array      myVar      myvar      _subroutine   my_var
x1         x348
```

Reserved Words

Reserved words are identifiers that have special meaning in C. Unless a reserved word is redefined as a preprocessor macro, it can only be used for the meaning that C assigns to it, except that reserved words can appear in comments or string constants. The reserved words in C areshown below. The reserved words that are underlined are reserved in ORCA/C, but not in ANSI C.

```
auto       asm        break      case       char       comp       const
continue   default    do         double     else       enum       extended
extern     float      for        goto       if         inline     int
long       pascal     register   return     segment    short      signed
sizeof     static     struct     switch     typedef    union      unsigned
void       volatile   while
```

Reserved Symbols

Reserved symbols are the punctuation of the C language. Reserved symbols are used as mathematical operators, for forming array subscripts and parameter lists, for separating statements, and so forth. With some restrictions, reserved symbols can also be used in comments, string constants, and characters constants. See the sections below for details.

The reserved symbols in C are:

```
!       %       ^       &       *       -       +       =       ~       |
.       <       >       /       ?       ->      ++      --      <<      >>
<=      >=      ==      !=      &&      ||      +=      -=      *=      /=
%=      <<=     >>=     &=      ^=      |=      (       )       [       ]
{       }       ,       ;       :
```

In some older versions of C, the assignment operators (like +=) could be specified with the equal sign first, as in =+. This caused some semantic problems in the language; most modern compilers, including ORCA/C, do not permit the equal sign to come first. Some compilers also permit white space to appear between the characters, as in + =. ORCA/C does not allow this, either.

Some older computers do not support complete character sets on their terminals or printers. For that reason, the C language includes trigraphs. A trigraph is a sequence of three characters, two question marks and a third character, which can replace another character. While there is no reason to use trigraphs on the Apple IIGS, you should be aware of their existence, since they can occasionally cause problems with string constants and character constants. The trigraphs, along with the character they represent, are shown below.

trigraph	character
??(	[
??)	]
??<	{
??>	}
??/	\
??!	|
??'	^
??-	~
??=	#

For example, the following two lines are completely equivalent in C.

```
while (~one | theOther) printf("Huh|\n");
while (??- one ??! theOther) printf("Huh??!\n");
```

Note the trigraph that appears in the string. This is one place where trigraphs can cause problems. The idea in the example shown was probably to write two question marks and an exclamation point, but this was translated into a single vertical bar. To avoid this, you can use \? in the string constant to represent one of the question marks, as in

```
while (~one | theOther) printf("Huh?\?!\n");
```

Continuation Lines

If a back slash character (\) appears as the last character on a line, the line is continued from the first character of the next line. Lines may be continued in the preprocessor, in string constants, between any two tokens, or even in the middle of a token.

See the section discussing string constants for details on continuing strings. See Chapter 12 for details on using continuation lines in the preprocessor.

Constants

Constants are used to place numbers and strings into the source code of the program. Each kind of constant has its own unique format, so they are discussed separately.

Decimal Integers

Decimal integers come in two sizes and two kinds. The two sizes are referred to as integer and long integer, while the two kinds are signed and unsigned.

Signed integers consist of one to five digits. The number represented must range from 0 to 32767, and with the exception of the number 0, no number can start with a leading zero. (See octal integers, below, for the reason for this restriction.) You may use a leading - character to form a negative number, although the - character and the number are technically two separate tokens. In practice, this technical distinction is rarely important.

If the number exceeds 32767, or if it is followed by an l or L, the number becomes a long integer. Long integers can range from 0L to 2147483647L. Please note that the l or L character must follow immediately after the last digit, with no intervening white space characters.

Unsigned integers are integers which are followed by a u or U. Unsigned integers can range from 0U to 65535U. Unsigned integers have the same effect on the type of an expression as an unsigned variable; see Chapter 17 for details.

Unsigned long integers include any unsigned integer larger than 65535U, as well as any unsigned integer with an l or L appearing either before or after the u or U. Unsigned long integers can range from 0UL to 4294967295UL. As with unsigned integers, using an unsigned long integer constant will affect the type of the expression.

The table below shows some examples of legal decimal constants.

integer	unsigned integer	long integer	unsigned long integer
`0`	`0U`	`0L`	`0LU`
`35`	`35u`	`35l`	`35ul`
`32767`	`65535u`	`100000`	`100000u`
`600`	`600U`	`2147483647l`	`4294967295ul`

Octal Integers

Octal numbers are integers represented in base eight, rather than the more familiar base ten. Octal numbers are made up of the digits 0 to 7. In C, octal numbers are distinguished from decimal numbers by a leading zero. Any number whose first character is zero is interpreted as an octal number in C, and may not contain the digits 8 or 9. This can lead to unexpected results if you are not aware of this convention. For example, the statement

```
printf("%d\n", 010);
```

will print 8 to the screen, since 010 is 8 in base 8.

As with decimal integers, octal integers can be long or short, signed or unsigned. The range of integers allowed in each representation are show below.

```
int                 0            077777
unsigned int        0            0177777u
long                0            017777777777L
unsigned long       0            037777777777uL
```

One point of confusion this often leads to is the base of the number 0. Technically, the number 0 is an octal number, but in practice it doesn't make any difference. You may use it when a decimal number is required because the difference in base only matters when the number is converted from the string you type in when you enter a program to the internal number used in calculations. In practice, then, you can use 0 as either a base 8, base 10 or base 16 number.

The table below shows some examples of legal octal constants, with their decimal equivalents.

octal integer	decimal integer
0	0
07	7
010	8
0100	64
077777	32767
0177777u	65535u
017777777777L	2147483647L
037777777777uL	4294967295ul

Hexadecimal Integers

Hexadecimal numbers are integers represented in base sixteen, rather than the more familiar base ten. Hexadecimal numbers are made up of the digits 0 to 9 and the letters a to f or A to F. In C, hexadecimal numbers are distinguished from decimal numbers and octal numbers by a leading zero, followed by an x character or X character.

As with decimal and octal integers, hexadecimal integers can be long or short, signed or unsigned. The range of integers allowed in each representation are show below.

```
int                 0            0x7FFF
unsigned int        0            0xFFFFu
long                0            0x7FFFFFFFL
unsigned long       0            0xFFFFFFFFuL
```

The table below shows some examples of legal hexadecimal constants, with their decimal equivalents.

hexadecimal integer	decimal integer
`0x0`	`0`
`0X7`	`7`
`0x10`	`16`
`0x100`	`256`
`0xa`	`10`
`0xF`	`15`
`0x7FFF`	`32767`
`0xFFFFu`	`65535u`
`0x7FFFFFFFL`	`2147483647L`
`0xFFFFFFFFuL`	`4294967295ul`

Character Constants

Character constants are formed by enclosing a single character in quote marks, as in

```
'a'
```

Character constants are treated exactly as if an integer constant equivalent to the ordinal value of the character was used instead of the character constant. For example, in any program that contains the character constant 'a', it would be legal, and have no effect on the executable program, to replace the character constant with 97.

That certainly doesn't mean there is no need for character constants. If you are trying to write a portable program that is comparing against the character a, you should use a character constant, since not all computers use the integer value 97 for a lowercase a.

ORCA/C uses the ASCII character set to determine the ordinal values of the characters.

Some characters in the ASCII character set cannot be represented using a key that can be typed from the keyboard. These characters are represented as escape sequences, and are discussed later in this chapter.

The C language does not specify if characters are signed or unsigned; that detail is left up to the implementor of the compiler. In ORCA/C, characters constants are signed.

Examples:

'a'   'A'   '\''   '"'   '\040'   '\40'   '\x10'

String Constants

String constants consist of any sequence of characters except new line or the quote character, enclosed in quote characters. (Both the new line character and the quote character can, however, be represented as an escape sequence.) As with character constants, escape sequences are used to represent non-printing characters; they are described in the next section.

Internally, strings are represented as a sequence of bytes, one for each character in the string, followed by a terminating null byte. (A null byte has a value of zero.) The value of each byte is the ordinal value for the character, as specified by the ASCII character set, described in Appendix D.

Unlike other constants, using a string constant in an expression does not cause the compiler to load the value of the string. Instead, as with arrays, the address of the first character is loaded. In practice, this means that you can use a string constant wherever a pointer to a string is required, as in the famous Hello, World program, shown below.

```
int main(void)

{
printf("Hello, world.\n");
}
```

String constants can be spread across more than one line. There are two ways to do this. The first method has been a part of the C language for quite some time. It involves using the \ character as the last character on the line. In that case, the string constant continues with the first character of the next line. The second method is new to ANSI C. In the second method, you simply code two string constants with no intervening tokens, and the compiler plugs them together. The Hello, World program is shown below using these two methods.

```
int main(void)

{
printf("Hello, \
world.\n");
}


int main(void)

{
printf("Hello, "
        "world.\n");
}
```

Note that the second method allows you to put spaces or tabs in the program to improve readability, while the older method of using a continuation line requires that the continuation of the string start in column 1.

ORCA/C imposes two limits on strings. First, no single string constant may have more than 4000 characters. Second, no single function can have string constants whose total length exceeds 8000 characters.

Examples:

```
"Hello, world.\n"
"This is a string constant"   " that has been broken into two parts."
"He said, \"I have arrived.\""
"What?\?!"
```

Escape Sequences

Escape sequences are used in string constants and character constants to represent characters that would otherwise be difficult to type from the keyboard, or that interfere with the construction of the constant itself. Escape sequences consist of the \ character followed by a single character, an octal constant, or a hexadecimal constant. (In escape sequences, hexadecimal constants must begin with a lowercase x.) The table below shows the escape sequences that consist of a single character, along with the equivalent integral value and the standard use for the escape sequence in C.

escape sequence	integral value	meaning
a	7	alarm (bell)
b	8	back space
f	12	form feed
n	10	new line
p	(special)	p-string
r	13	carriage return
t	9	horizontal tab
v	11	vertical tab
\	92	\ character
'	96	' character
"	34	" character
?	63	? character

Four of these escape sequences are used because of the syntax of character constants and string constants. Since the back slash character is used to start an escape sequence, you can also follow it with a second back slash character to place a single back slash character in a constant. The quote mark (") ends a string constant, and the single quote mark (') ends a character constant, so both have an escape sequence to allow you to put these characters in string and character constants. The ? character is used when a ? in a string or character constant might be confused with a trigraph. All of the remaining escape sequences except \p are used to represent control characters that have special meaning to the console driver. While the value used for each of these control characters will vary from computer to computer, they always perform the same action when the string or character constant is used with the standard output libraries.

The escape sequence \p is not used in standard C. It is used in ORCA/C to allow the formation of a p-string. Normally, strings end with a terminating null character. Unfortunately, many of the tool calls in the Apple IIGS toolbox require a p-string, which uses a length byte to indicate the length of a string. When you need to specify a p-string as a constant, use the escape sequence \p right after the opening quote mark. The final string will start with a byte that indicates how many characters are in the string, and will be followed by a normal C string. The terminating null character is not counted as one of the characters when determining the value of the length byte. The \p escape sequence is treated as the character p in a character constant, as well as in a string constant if the escape sequence is not the first character in the string.

When you need to specify a specific numeric value in a character or string constant, you can follow the back slash character by one to three digits, which are then interpreted as an octal number. For example, since ORCA/C uses the ASCII character set, which specifies an ordinal value of 041 octal (33 decimal) for the character !, the character constants '!' and '\41' represent the same value. You can also use a lowercase x followed by one , two or three digits to represent the value in hexadecimal notation. Thus, '\x21' is also a valid way to represent '!'. You should be careful when using values in strings. While '\41' represents the ! character in a character constant, "\410" is not the same as "!0", as you might intend. Instead, the finished string has a single character with an ordinal value of eight. (The most significant two bits do not fit into a byte, and are discarded.)

The C language does not specify what the compiler will do if you follow the \ character by a character other than one of the ones discussed. In ORCA/C, use of a non-escape character after the \ character will place that character in the constant. For example, the character constants `'\g'` and `'g'` are exactly the same.

For examples of strings and character constants that include escape sequences, see the two preceding sections.

Real Numbers

Floating-point constants are used to represent numbers that do not have integral value, or that cannot be represented using an integer because they are too large or too small. The general format is a sequence of digits, followed by a decimal point, followed by another sequence of digits, and an exponent, as in

```
3.14159e-14
```

The exponent can start with either an uppercase E, or a lowercase e, as shown.

The format for floating-point constants can vary quite a bit from this general form. You can leave out the digit sequence before or after the decimal point, as in `1.e10` or `.1e10`. In fact, you can leave off the exponent, too, as in `1.` or `.1`. You must have either an exponent or a decimal point, but if you specify an exponent, you can omit the decimal point, as in `12e40`.

The exponent, if specified, starts with either a lowercase or uppercase e. This is followed by an optional plus or minus sign, and then by a sequence of one or more digits. The number is represented internally as an extended SANE format number, giving an accuracy of a little over 19 decimal digits, and an exponent range of -4932 to 4932. Keep in mind that if the number is stored in a float or double variable, the accuracy will be reduced to match the accuracy of the variable.

C allows a floating point constant to be followed by f or F to indicate a float value, or l or L to indicate long double. While these characters are allowed, they have no effect on a floating-point constant in ORCA/C.

White Space

White space characters may be used to separate the tokens in a program. White space characters include the space, line feed, carriage return, vertical tab, horizontal tab, back space, and form feed characters. The line feed, carriage return, form feed and vertical tab characters all end the current line, which has special meaning in the preprocessor and when continuing lines. With that exception, replacing any sequence of white space characters outside a string or character constant with a single space has no effect on the finished program. Generally, white space characters are used to format the program, making it easier to read.

Comments

A comment starts with the characters /*, and ends with the first occurrence of the characters */. Comments are used solely to document the source code; replacing a comment with a single space will have no effect on the finished program.

Note that comments can extend over more than one line, and that they can be used in preprocessor commands.

C does not allow recursive comments (sometimes called nested comments); that is,

```
/*   /*   */   */
```

is not a legal comment in C. You can use the preprocessor's #if, #endif commands to cause the compiler to ignore large sections of code which may include comments.

Chapter 12 – The Preprocessor

The C preprocessor is a series of commands that tell the compiler to take certain actions. Using the preprocessor, you can tell the compiler to skip certain sections of source code, to replace some source code by other source code, and so forth. While the preprocessor is built right into ORCA/C, logically, preprocessing occurs before the program is compiled.

Syntax

Commands to the preprocessor all start with the # character, which must appear before any other non-white space character in the line. This is followed by the preprocessor command, which may require other parameters. You can separate the preprocessor command from the # character with white space characters. The preprocessor command ends with the first new line, vertical tab, carriage return, or form feed character. The # character can appear on a line by itself, in which case it is ignored.

The ability to place white space characters before the # character and between the # character and the name of the preprocessor command is a recent addition to the C language. If you are concerned with portability, you may wish to code the preprocessor commands as we do in the examples, without white space characters.

There are two ways to extend a preprocessor command over more than one line in the source file. The first is to continue the line by placing a \ character at the end of the line, as in

```
#define sec(x)  \
          (1.0 / cos(x))
```

The other way to extend a preprocessor command over more than one line is to start a comment on one line, and finish it on another line, as in

```
#define                        /* trig extensions
          */     sec(x)   (1.0 / cos(x))
```

While the preprocessor does not parse the source file, it does break the source file up into tokens to find macro names. The same rules are shared by the preprocessor and the compiler for forming tokens and comments.

Preprocessor Macros

The #define preprocessor command allows you to define macros. When the macro is used later in the source code of the program, it is replaced by the tokens layed out in the #define command. One of the most common uses of preprocessor macros is to define constants using simple textual replacement. For example, you can define the boolean constants true and false, as shown in the simple program below.

```
#define TRUE -1
#define FALSE 0

int main(void)

{
int bool;

bool = TRUE;
if (bool)
   printf("Hello, world.\n");
}
```

While the compiler uses a more efficient mechanism to implement preprocessor macros, the way to think of this program is that the two #define statements define two macros, called TRUE and FALSE. Whenever the preprocessor finds one of those words in the program, it is replaced by the text that follows the words. In our example, then, preprocessing occurs before the compiler starts to compile the program, so the program the compiler actually compiles looks like this:

```
int main(void)

{
int bool;

bool = -1;
if (bool)
   printf("Hello, world.\n");
}
```

Before moving on to more complicated examples, a few points are worth mentioning. First, the text that will replace the name of the macro starts with the first non-white space character after the name of the macro itself. This text is converted into a stream of tokens by the preprocessor, and stored as tokens. This has an important consequence. At first glance, it might seem that a statement like

```
bool = 0FALSE;
```

would be legal, since, after macro replacement, the statement looks like this:

```
bool = 00;
```

In fact, the statement is not legal, because the macro has already been converted into an integer with a value of zero, so the compiler sees two integer constants after the equal sign. The correct equivalent, then, is

```
bool = 0 0;
```

One thing you will notice in all of the examples is that all of the macro names are capitalized. You are not required to capitalize the names of macros, but this is a common convention used by many C programmers, and we will follow that convention in our examples. It is worth pointing out, though, that the names of macros, like all identifiers in C, are case sensitive.

Macros are not limited to simple textual replacement. For example, you can define a trigonometric function for the secant using a macro.

```
#define sec(x) (1/cos(x))
```

Note that in this case, since we want the macro to blend in with the standard C math library, we have used lowercase letters in the name of the macro.

When you use a macro that has parameters in a program, you can substitute any number of tokens for the parameter. For example, all of the following are legal uses of the macro preprocessor, although not all of the examples result in legal C programs.

```
sec(pi)
sec(0.45)
sec(pi/12.0)
sec("strings are allowed, although this example will not compile")
sec((pi/12.0+.45)*0.01)
```

Macros can have any number of parameters. To create a macro with more than one parameter, list the names of the parameters separated by commas. For example, you could define a macro to find the distance between two points, as in the following small program, which also shows one macro calling another.

```
#include <math.h>
#include <stdio.h>

#define sqr(x) ((x)*(x))
#define distance(p1,p2) (sqrt(sqr(p1.x-p2.x)+sqr(p1.y-p2.y)))

int main(void)

{
struct point {float x,y;} point1,point2;

point1.x = 1.0;
point1.y = 1.0;
point2.x = 3.0;
point2.y = 1.0;

printf("The distance is %f\n", distance(point1,point2));
}
```

There are a few important points to note about the syntax of macros that have parameters. First, the opening parenthesis must appear immediately after the name of the macro in both the definition and the use. This is to prevent confusion between a macro parameter and a simple textual replacement macro whose first character is a left parenthesis. Second, you can use parentheses within the macro call to enclose token streams that include commas, so long as the parentheses are legal in the C program produced by the macro expansion. The names of macro parameters, like macros, follow the same rules as identifiers in C, with the exception that reserved words are not reserved in the preprocessor. In fact, it is legal, although very poor form, to define a reserved word as a macro.

It is not legal to define more that one macro using the same name.

Within a macro, it is possible to merge the string value of a macro parameter with adjacent string constant, or to treat the parameter as a string constant. The # operator, when it appears before the name of a macro parameter, indicates that the parameter is to be treated as a string rather than a token. For example, the following code fragment will print the familiar Hello, world. string to the screen.

```
#define greet(who) "Hello, " #who "."

printf(greet(world));
```

The macro preprocessor can also merge two tokens to form a new token. The ## operator controls this process. The following code fragment is equivalent to strPtr = setPtr.

```
#define ptr(where) where ## Ptr

ptr(str) = ptr(set)
```

Macros can also be removed with #undef. #undef takes a single parameter, which is the name of a macro to undefine. The macor by the given name is removed from the preprocessor's macro symbol table. It is not an error to undefine a macro that has never been defined. Once the macro has been undefined, it is also not an error to redefine it.

```
#undef greet
```

There are several predefined macros that exist in any C program. These macros can be used in your program, but you cannot change them or remove them via the #undef command. Each of the predefined macros places a single token in your program; the token, and what it means, is shown in the table below.

macro	token	use
__DATE__	string	The date as of the start of the compilation, in the form Mmm dd yyyy, e.g., "Jan 12 1989".
__FILE__	string	Name of the current source file.
__LINE__	integer constant	Line number of the line being compiled. Each physical line in the file is counted, even if the line is in a comment or is skipped because of preprocessor commands.
__TIME__	string	Time as of the start of the compilation, in the form hh:mm:ss, e.g., "15:36:12".
__STDC__	integer constant	A non-zero value in any compiler that implements ANSI C. In ORCA/C, the value is -1. In a compiler that does not implement ANSI C, this macro will not be defined.
__ORCAC__	integer constant	ORCA/C will return a -1. In any other C compiler, this macro will not be defined.
__VERSION__	string	The compiler version in the form "2.0.0".

Entering Macros from the Command Line

In addition to defining macros in the source file with the #define command, you can also define macros from outside of the compiler when you compile the program. ORCA/C now allows definition of macros from the command line. The macros are defined within a language dependent parameter. For example,

```
run prog.cc cc=(-dPARM)
```

defines a macro named PARM.

There are basically two ways to define a macro from the command line. The definition shown above has the form

```
-d<name>
```

which is exactly equivalent to

```
#define <name> 1
```

at the beginning of the source file. This defines a macro with the name given, and assigns the value 1. You can also define a macro with the form

```
-d<name>=<token>
```

where token can be any integer, long integer, floating-point constant, or identifier. This is exactly equivalent to

```
#define <name> <token>
```

at the beginning of the source file.

No blanks are allowed in the macro definition. Spaces can appear before or after the definitions, and are used to separate multiple definitions.

These definitions can be used with the COMPILE, CMPL, and RUN command, as well as with all of the aliases for these commands. The definitions apply to all source files specified in the command, not just the first one. You can also enter these definitions from the "Language Parms" field of the Compile dialog from the desktop development environment. The definitions are ignored by languages other than the C compiler.

Here are more examples of macro definitions from the command line.

```
run prog.cc cc=(-DTEST -DDEBUG=TRUE)
compile file1.cc file2.cc cc=(-dday=14 -dmonth=September -dyear=89)
cmpl cc=(-dPI=3.14159e+01)
```

Including Files

The #include command is used to deal with situations where a source program is made up of more than one source file. The most common use of the #include directive is to include interface files for the standard C libraries. For example, to be completely correct, the common Hello, World program should actually start off with a #include command that includes the header for the standard input and output library, like this:

```
#include <stdio.h>

int main(void)

{
printf("Hello, world.\n");
}
```

The include statement is followed by a file name, which can be enclosed in quote marks or brackets. The compiler compiles all of the lines in the included file, then returns to the file that contained the #include command, and continues compiling with the line immediately after the #include command. In the example, the name of the header file for the standard input and output libraries is stdio.h. This file can be found with all of the other standard header files, which include not only the standard C libraries, but also the Apple IIGS toolbox interface. These header files are located in the ORCACDefs folder, which, in turn, is located in the LIBRARIES folder. Any time you enclose a file name in brackets, the compiler will look for the file in the ORCACDefs folder.

The other common use for the #include command is to include source code you have written specifically for a program. This could include custom header files, macro definitions used in more than one source file, or even a single program which has grown too large to edit. In this case, you would enclose the file name in quote marks, as in

```
#include "mymacros"
```

When the file name is enclosed in quote marks, the compiler looks for the file in the current directory. If the file you want to include is located in a folder in your current directory, you can use a partial path name; you can also use a full path name if the file you want to include is located on another disk, or in some other location on your current disk. Examples are shown below.

```
#include "/network/project.x/secret.macros"
#include "macrofolder/macro.file.1"
```

It is possible to use a macro for the file name, provided the file name can be broken down into tokens. In the case of a quoted file name, this will always work, since the file name is a string constant. In the case of a file name enclosed in brackets, you will be able to use any GS/OS file name, but some file names from other file systems may not work. Note that the ability to use a macro for the file name is a recent addition to the C language, and may not be present in all C compilers.

There is no fixed limit to the nesting level for included files. For example, an included file can include another file, which can also include still another file, and so on. This process can continue for as long as memory is available.

ORCA/C supports another file inclusion mechanism, the #append command. This command is used to chain two files together. Like the #include command, the #append command requires a file name as a parameter, and this file name can be enclosed in brackets or quote marks. Brackets are still used to indicate that the file should come from the ORCACDefs folder, and quote marks are used for files in the local directory, or for full path names. You can also use a macro to specify the path name, with the same restrictions that applied to the #include command.

There are two principle differences between the #include command and the #append command. The first is that all lines after the #append directive are ignored. Once the compiler starts processing lines in the appended file, it never returns to the original file, as the #include directive does. Conceptually, the #append command is like a goto, while the #include command is like a subroutine call. The other difference between the two commands is that, if the #append command appears in the top level file, the file that is appended can be a source file for some language other than C. The top level file is the source file you actually compile, that is, a file that has not been included using the #include command. This powerful feature means that you can create a program using two or more languages, and then compile the entire program with a single step.

Precompiled Headers

Many C programs, especially those that use the Apple IIGS toolbox, start with a list of header files to include. In practice, compiling the header files can actually take longer than compiling the executable part of the program.

ORCA/C 2.0 tries to cut compile times by eliminating the need to compile the headers files over and over. As each header file is compiled, ORCA/C writes the new symbol table information to a file we will call the .sym file. This file is in the same folder as the initial source file. The name of the file is formed by removing the last extension on the source file name, if any, and adding the characters ".sym". For example, the .sym file for a source file named foo.cc would be foo.sym. The next time the file is compiled, ORCA/C can read the symbol table from the .sym file, often cutting compile times more than in half.

There are many ways to implement this concept. Two design considerations drove our choice for the implementation method. First, we wanted a mechanism that was 100% transparent to anyone using the compiler. With the exception that you will see .sym files formed for each compile, ORCA/C precompiled headers are completely automatic. The other factor that drove the design is that macro definitions that precede include statements can, and often do, effect the way a header file is compiled. For example, it is very common to use a macro to override the size of an array in a tool header file. This useful feature of the C language makes it impossible to compile the header file itself, replacing it with a symbol table, without requiring programmers to know when forcing a recompile of the symbol tables is appropriate.

ORCA/C does several things to determine when a .sym file must be rebuilt. First, if the source file changes in any way before the include statements, the symbol file is rebuilt. If the time or date stamp on any include file changes, the symbol file will also be rebuilt. Finally, if an include file is missing or cannot be accessed, the .sym file is rebuilt.

There are also some restrictions that apply to which header files are included in the .sym file. The first code-generating function or initialized variable in the program completes the .sym file, and all subsequent header files are compiled in the normal way. For example, if an include appears after a function definition (not declaration) it will not be included in the .sym file. In addition, any header file that contains an initialized variable or function definition will not be included in the .sym file, and will block subsequent header files, too.

Assuming the .sym file must be rebuilt, it actually takes two compiles before the process is complete. On the compile where the compiler determines that the .sym file should be rebuilt, it uses any information from the .sym file up to the point of change, then deletes the old .sym file. The new .sym file is built on the next compile.

There are two flags that control the use of .sym files. On any of the compile commands, the i flag tells the compiler to compile the program as if the compiler did not support precompiled headers. Any .sym file that is present is ignored, and a new .sym file will not be built. The -r flag forces the compiler to rebuild the .sym file, even if the compiler does not see any reason to do so.

Precompiled headers are not supported by the small memory version of the compiler.

Conditional Compilation

The preprocessor includes a powerful conditional compilation mechanism. Using conditional compilation, it is possible to control the way a program is compiled by making small changes in the source for a program.

The two commands which form the cornerstone for conditional compilation are #if and #endif. The #if command has an expression for an operand. This expression is evaluated. If the result of the expression is zero, all of the lines from the #if command to the matching #endif command are skipped by the compiler. If the result of the expression is not zero, the lines following the #if command are compiled in the normal way, and the matching #endif command is ignored.

As an example, we could write a very short program to test to see if a compiler implements ANSI C. All ANSI C compilers define the __STDC__ macros as a non-zero value. In compilers that do not implement ANSI C, the macro will not be defined. When an undefined macro name is used in an expression in an #if command, it is replaced by the integer constant 0. Our program, then, looks like this:

```
int main(void)

{
#if __STDC__
printf("This compiler implements ANSI C.\n");
#endif
}
```

While the lines between the #if and #endif statements are skipped if the expression in the #if statement evaluates to 0, the lines skipped must still be made up of legal C tokens. For example,

```
#if 0
@
#endif
```

is not legal, since the character @ is not a legal C token.

#if statements may be nested. For example, we could expand our previous example to see if the compiler being used is ORCA/C, like this:

```
int main(void)

{
#if __STDC__
   printf("This compiler implements ANSI C.\n");
   #if __ORCAC__
      printf("In fact, this is ORCA/C.\n");
   #endif
#endif
}
```

With a few exceptions, the rules for forming an expression in an #if statement are the same as the rules for expressions in the C language. The exceptions have to do with the fact that the expression is evaluated at compile time, rather than at run time, and also with features have been added to simplify tasks in the preprocessor. The major difference between expressions used in the #if command and expressions used in the program is that the expression that appears after the #if command must be a constant expression. For a detailed discussion of constant expressions, see Chapter 17.

Preprocessor expressions can make use of two features which cannot be used in any other expression. The first has already been mentioned: when an undefined macro is encountered, it is given a value of 0. The second feature is the defined operator. The defined operator is followed by the name of a macro, which may be enclosed in parentheses. The result of the operator is 1 if the macro is defined, and 0 if it is not defined. Rewritten to use the defined operator, our example program that determines if a compiler is an ANSI C compiler looks like this:

```
int main(void)

{
#if defined __STDC__
printf("This compiler implements ANSI C.\n");
#endif
}
```

As with the C language, the preprocessor's #if statement can have an else clause. Any #if statement can have a #else statement between the #if statement and the matching #endif statement. We can use this feature to extend our test program so that it prints a message if the compiler is not an ANSI C compiler, too.

```
int main(void)

{
#if __STDC__
printf("This compiler implements ANSI C.\n");
#else
printf("This compiler does not implement ANSI C.\n");
#endif
}
```

Complex conditions can be handled using the #elif command. The #elif command is a combination of the #if command and the #else command. Like the #else command, the #elif command is used after a #if command, but before the matching #endif command. It must also come before the #else command if there is one. Like the #if command, the #elif command is followed by an expression. This expression is only evaluated if the previous #if command or #elif command evaluated to zero. In that case, the expression is evaluated. If the result is zero, the preprocessor scans forward to the next matching #elif, #else or #endif command, and takes appropriate action. If the result of the expression is not zero, all statements up to the next matching #elif command, #else command, or #endif command are compiled. Any lines up to the matching #endif statement and the #endif statement itself are then skipped by the compiler. Note that more than one #elif command can appear in a single #if structure, in which case the preprocessor evaluates the expression in each statement in turn until one of the expressions results in a non-zero value. The lines following that statement are then compiled. If all of the #if and #elif statements have expressions that evaluates to zero, the lines following the #else statement are compiled.

As an example, we can expand the program that determines what compiler we are using to detect ORCA/C or APW C.

```
int main(void)

{
#if __ORCAC__
printf("This compiler is ORCA/C, which implements ANSI C.\n");
#elif __STDC__
printf("This compiler implements ANSI C.\n");
#elif __APWC__
printf("This compiler is APW C, which does not implement ANSI C.\n");
#else
printf("This compiler does not implement ANSI C.\n");
#endif
}
```

There are two special forms of the #if statement which are used to test for the existence of a macro. They are #ifdef and #ifndef. Both statements require an operand that is a single identifier. If the identifier has been defined as a preprocessor macro, then #ifdef works exactly like #if 1, and #ifndef works like #if 0. If the identifier is not a preprocessor macro, the results are switched. Recoding our program to determine if a compiler is an ANSI C compiler to use these commands, we have the following:

```
int main(void)

{
#ifdef __STDC__
printf("This compiler implements ANSI C.\n");
#endif
#ifndef __STDC__
printf("This compiler does not implement ANSI C.\n");
#endif
}
```

Line Numbers

Historically, the C language has been associated with preprocessors. The preprocessor described in this chapter was originally a separate program from the C compiler. It took a source file as input, and produced another source file as output; the output file was then compiled. The C++ language is a more recent example of a preprocessor used with the C language. There are also numerous examples of cross-compilers that work by taking a program written in one language, say FORTRAN, and producing a program written in C for compilation by the C compiler.

This presents a problem when the compiler needs to flag an error. Meaningful error messages are associated with information about the source file in which the error occurs, and the line where the error occurs. It is very discouraging, for example, when a compiler is only able to tell you that a divide by zero error occurred somewhere in your 10,000 line program; it is far more useful if the compiler identifies which line of which source file the error occurred in.

The #line command is used by preprocessors for this purpose. It takes one or two parameters, either of which may be produced by expanding a macro. The first, which is required, is an integer constant. The value is used by the compiler as the line number of the next line that is compiled. The second parameter, which is optional, is the name of the source file, enclosed in quote marks. This name is used until it is overridden by another #line command. The line number and file name specified this way are used by the compiler for compile-time errors, reported by the program when a run-time error occurs and the #pragma debug directive has been set to an appropriate value, and used by the source-level debugger to show where you are in the original source file.

Flagging Errors

The #error statement is used to produce a compile-time error message. It uses a single parameter, which must be a string constant. This string is printed as an error message at compile time. For example, the program

```
int main(void)

{
#ifndef __GARBONZOC__
#error "This program requires Garbonzo C."
#endif
}
```

would generate the compile-time error message

```
#error: This program requires Garbonzo C.
```

just before the #error command in the output listing.

Pragmas

The #pragma command is used for preprocessor commands that are specific to a particular compiler. Each #pragma command that ORCA/C recognizes is followed by an identifier that specifies the type of the statement. Any #pragma statement that does not start with one of the identifiers described below is skipped.

cda

```
#pragma cda name start shutdown
```

The cda command is used to create a classic desk accessory. Classic desk accessories are not executed like normal C programs; instead, they are copied into the DESK.ACCS folder in the

SYSTEM folder, where they can be used from any program that follows standard rules for the Apple IIGS.

Normal C programs must contain a function called main. This function is the one executed when the program starts. Classic desk accessories do not have to have a function called main. Instead, you specify the name of the function that will be called when the classic desk accessory is executed as the *start* parameter to the cda command.

When the operating system is shutting down, it will call each of the classic desk accessories to give it a chance to do any shut-down processing that it may require. Each classic desk accessory must have a shut-down function. The *shutdown* parameter is the name of your shut-down function. If you do not need to do any shut-down processing, you must still have a shut-down function, but it can be an empty function.

Each classic desk accessory has a name. This name appears in the CDA menu, which is used to select the classic desk accessory to execute. The *name* parameter is used to specify the name of your classic desk accessory. It must be a string constant.

The cda command must be used before the start of the first function in the program. If it is used, you cannot use another cda command or an nda command.

cdev

```
#pragma cdev start
```

The cda command is used to create a Control Panel Device (CDev) for the Control Panel from System 6.0. Classic desk accessories are not executed like normal C programs; instead, they are copied into the CDEV folder in the SYSTEM folder, where they can be used from within Apple's Control Panel.

Normal C programs must contain a function called main. This function is the one executed when the program starts. CDevs do not have to have a function called main. Instead, you specify the name of the function that will be called when the classic desk accessory is executed as the *start* parameter to the cdev command. The function itself takes two long integer parameters and an integer parameter, returning a long integer.

For a complete description of Control Panel Devices, as well as details about the parameters passed to this function and the value returned by the function, see Apple II File Type Notes for file type $C7 (CDV).

For a short example of a CDev, see the :CC.Samples: disk.

databank

```
#pragma databank parm
```

There are several instances that arise when using the toolbox where you need to create a function that a tool will call. When the C compiler creates the code for a function, it assumes that the data bank register is pointing to the bank that contains the global scalars. When a function is called by a tool, there is no guarantee that this is true. To solve this problem, you must use the databank directive before any function that will be called from a tool. This directive tells the compiler to save the original data bank register, and then set the data bank register to point to the bank containing the global scalars. Before returning from the subroutine, the original data bank register is restored.

```
    #pragma databank 1
```

Immediately after the function, switch back to the normal calling conventions using the command

```
    #pragma databank 0
```

Functions that have data bank restoration on can still be called from other C functions, but they will be slightly less efficient than functions that do not have data bank restoration enabled.

See also the toolparms directive.

debug

```
#pragma debug parm
```

The debug command is used to control the types of debugging the compiler will do. The various debugging features can be very useful when you are developing a program, but each debugger feature requires run-time code which takes up space and time in your finished program.

This command is only used in rare cases. If you are using the desktop development environment, you will normally use the debug check box in the compile dialog to control debugging. When that box is checked, all of the debugging features described here are enabled. When that box is not checked, all of the debugging features are disabled.

The debug command requires a single integer parameter. This parameter is actually a series of flags. For each flag, a value of 1 turns the debugging feature on, and a value of 0 turns it off. Even if new debugger features are added in the future, then, the command

```
    #pragma debug -1
```

will enable all debugging features, while the command

```
    #pragma debug 0
```

will turn all debugger features off.

Setting bit 0 (a value of 1) turns range checking on. The only checking this enables in the C compiler is a check for stack overflows. This check is made at the start of each function. The check ensures that there is enough room left on the run-time stack to declare all of the local variables needed by the function. This reduces the chance of crashes due to stack overflows. It is still possible to overflow the stack with a very complicated expression or by calling a tool.

Setting bit 1 (a value of 2) tells the compiler to generate debug code for the source-level debugger. You should not set this bit unless the program will be executed from the desktop development environment. If you try to execute a program that has source-level debug code in it from the text shell or the Finder, the program will crash.

Setting bit 2 (a value of 4) tells the compiler to generate profile code. Profile code is used by the desktop development environment's profile command to tell you where the hot spots are in your program. You must set bit 1 if you set bit 2.

Setting bit 3 (a value of 8) tells the compile to generate trace back code. Without trace back code, a run-time error will simply report which error occurred. If you have enabled trace back code, you will get more information about the error. This information starts with the name of the function where the error occurred, along with the line number in the source file. This is followed by a table showing the function that called the one where the error occurred, and the line number the call was made from, then the function that called the previous one, and so forth, back to the function main. This information is written to error out, which defaults to the text screen in the text environment, and the shell window in the desktop environment.

Setting bit 4 (a value of 16) tells the compiler to check for stack errors. If this check is enabled, the compiler generates code for each function call that keeps track of the stack as the function is called, making sure the function removes exactly the right number of bytes from the stack. For example, if you call a function and pass an integer parameter, two bytes are pushed onto the stack. If the function expected a long integer, though, it will remove four bytes from the stack, and this error check would catch the error. When using this error flag, we suggest

```
#pragma debug 25
```

which also turns on stack overflow checking and tracebacks. With tracebacks enabled, the compiler doesn't just tell you that an error occurred, it also tells you where the error occurred, making the error a lot easier to track down and correct.

expand

```
#pragma expand int
```

The expand pragma allows you to see the tokens the compiler is actually compiling, essentially showing you the output from the preprocessor, with all of the preprocessor macros expanded. If INT is a non-zero value, the preprocessor prints the token stream sent to the compiler, after all tokens have been expanded. If INT is zero (the default), this information is not printed.   This feature is useful for debugging macros and examining the effects of character constants and escape codes. The exact expansions produced are:

1. All integers and character constants are expanded to base 10 values.
2. Escape sequences in strings are printed as hex escape sequences.
3. Macros are expanded.
4. Floating point constants are converted to exponential form.
5. Trigraphs are converted to their equivalent characters.
6. Preprocessor directives are removed from the source stream.

7. Any input skipped due to conditional compilation directives is removed from the source stream.
8. Comments are removed.

float

```
#pragma float card slot
```

By default, ORCA/C generates calls to the Standard Apple Numerics Environment (SANE) to perform floating-point calculations. If you have installed an Innovative Systems FPE card, however, ORCA/C can create a program that calls the card directly. To create a program that calls the FPE card, place the float directive before the first function in your program. The card parameter should be coded as 1. The slot parameter is no longer used, although a value must be coded If your program does not do any floating-point calculations, this directive will make no difference in the code or execution time. On some floating-point intense programs, however, the FPE card can speed up a program by a factor of 120.

The actual effect of this directive is to tell the compiler not to generate direct calls to SANE, forcing it to use library calls for all floating-point calculations.

In addition to using this directive, you should replace the SysFloat library in the Libraries folder with the library by the same name at the path :MoreExtras:FPE:SysFloat. This library generates calls to the FPE card, rather than calling SANE. Because of this arrangement, you will get some benefit from the FPE card by simply replacing the library, even if you forget to use this pragma.

If a program is compiled with the FPE libraries, an FPE card must be installed or the program will crash or give incorrect answers. If you must create a program that can work with or without an FPE card, write the program to make SANE calls, and use the SANE patches that come with the FPE card.

Card numbers other than 0 or 1 are not currently in use. They are reserved in case other floating-point cards are produced for the Apple IIGS.

ignore

```
#pragma ignore flags
```

ORCA/C is an ANSI C compiler, adhering to the language specification defined by the American National Standards Institute (ANSI). ANSI C is actually the latest major standard in a long line of languages that have used the general name C; there are, in fact, five major dialects of C and countless minor variations.

Judging from the changes made in ANSI C, one of the concerns of the standards committee was to make C a safer language, catching many errors in the compiler than might have resulted in incorrect programs in earlier compilers. In general, this is a very good goal, but some older C programs no longer compile under ANSI C compilers. The ignore statement tells ORCA/C to ignore certain kinds of checks that are required of ANSI C compilers so you can port older C programs a little easier.

The flags parameter is a series of bits, each controlling a specific error. If the bit is set, the error is *not* reported. The flags that are currently available are:

bit	use

0	If this bit is set, the compiler does not report illegal characters in the source stream when the characters occur in code that is not processed by the compiler. For example, it is fairly common for programs to use the $ character in file names, as in
	
	`#ifdef VAX`
	`#include <sys$stdio.h>`
	`#endif`
	
	This code is illegal under any properly implemented ANSI C compiler, whether or not VAX is defined. If bit 0 of the ignore pragma's flag word is set, and VAX is not defined, ORCA/C will not flag an error.

keep

```
#pragma keep name
```

The keep command sets the name of the output file. A name is provided automatically if you are using the desktop environment, although you can use this directive to choose your own name. The name you choose cannot be the name of an existing file unless the file is an executable file or object module.

The single parameter is the name of the output file. As with the parameter for the #include command, this name is enclosed in quote marks or brackets, indicating if the file should be placed in the current directory or the ORCACDefs directory. In general, you would not want to place the file in the ORCACDefs directory.

The keep command must be used before the start of the first function in the program, and only one keep command can be used in a program.

lint

```
#pragma lint int
```

The lint pragma forces stricter checking of programs than is required by the ANSI C standard. The checks are individually enabled and disabled by setting and clearing bits in int. To enable more than one check, add the values for the bits shown in the table below to form a single integer. To enable all lint checks, use a value of -1.

1 Flag the use of a function before the function is defined as an error. This is always bad form, but this flag is also very useful in checking to insure that all header files that should have been included have, in fact, been included. If you missed including the header file for string.h, for example, but used strcat in your program, the compiler will flag an error for using strcat.

Error message: "lint: undefined function"

2 Flag functions with no types as errors.

Error message: "lint: missing function type"

4 Flag functions with no prototyped parameter lists as errors. If you are using tool header files that do not have prototyped headers, but you still want to use this check for the rest of your program, put the lint pragma after the #include statements that include the tool header files.

Error message: "lint: missing parameter list"

8 Flag pragmas that are not recognized by ORCA/C as errors. Normally, ANSI C compilers ignore pragmas they do not recognize, assuming the source file is being used with more than one compiler. Because of this, spelling errors in a pragma can go unnoticed by the compiler; this bit helps find this sort of problem.

Error message: "lint: unknown pragma"

memorymodel

```
#pragma memorymodel parm
```

The parameter is an integer constant. If the value is zero, the compiler will generate code for the small memory model; a non-zero value will generate code for the large memory model. You should use a value of 1 for the large memory model to allow for future expansion of the number of memory models.

The large memory model is the most flexible. When you use the large memory model, your program can have up to 64K bytes of global variables other than arrays, structures and unions. Arrays, structures and unions can be as large as memory will allow, and you can have as many of them as will fit in memory. In particular, arrays are not limited to 64K bytes, nor is the total space used by arrays limited to 64K bytes. Dynamically allocated memory can also exceed 64K for a single chunk of memory.

When you use the small memory model, all global variables, including arrays, structures and unions, are limited to a single 64K byte area of memory. This area of memory is shared with any functions you use from the standard C library, and if you do not use the segment statement, with your program's code. In addition, the compiler assumes that you will not allocate any single array, structure or union that is larger than 64K bytes using the Memory Manager or malloc. This restriction applies to any single structure, not to the total amount of space in use; you can allocate arrays, structures and unions whose total space exceeds 64K. These restrictions allow the compiler to generate code that is much smaller and faster than the code generated when the large memory model is used.

It is possible to create programs that are much larger than 64K bytes in length without using the large memory model. The segment statement can be used to split a program into more than one executable code segment. The large memory model is only needed if the total global variable space exceeds 64K bytes, or if you will be creating and manipulating dynamically allocated structures, unions or arrays where a single structure, union or array exceeds 64K bytes. When you can use the segment directive instead of the large memory model, it is best to do that, since the compiler generates smaller, more efficient code in the small memory model.

Regardless of the memory model used, you can use the segment statement to place the code for the program in various static or dynamic segments. In addition, both memory models limit the run-time stack size. For information about the run-time stack, see the stacksize command, below.

nba

```
#pragma nba start
```

The nba command is used to create a HyperStudio New Button Action (NBA).

Normal C programs must contain a function called main. This function is the one executed when the program starts. NBAs do not have to have a function called main. Instead, you specify the name of the function that will be called when the NBA is executed as the *start* parameter to the nba command.

The function itself takes a single parameter of type `HSParamPtr` and returns void. When the NBA is called, HyperStudio passes a pointer to a structure containing a variety of information. Information is also returned in this structure.

HyperStudio supports a variety of callbacks; these are calls from the NBA back to HyperStudio to perform some action. From ORCA/C, callbacks are made using the `__NBACALLBACK` function, defined in HyperStudio.h. (Note: The name starts with two _ characters, not one.) This function requires two parameters: The callback number and a pointer to a structure like the one passed to the NBA. In most cases, you will actually pass back the same pointer passed to the NBA, but it is possible to make multiple copies of the structure. If you do make copies of the structure, though, be sure to initialize the copies carefully, generally by copying the original structure in it's entirety into the copy. There are several fields in the structure that must be initialized properly for a callback to work.

For a definition of the `HSParams` structure, along with a number of other useful declarations, see HyperStudio.h. For complete details on how to write HyperStudio NBAs, contact Roger Wagner Publishing.

For a short example of an NBA, see the :CC.Samples: disk.

nda

```
#pragma nda open close action init period eventmask menuLine
```

The nda command is used to create a new desk accessory. New desk accessories are not executed like normal C programs; instead, they are copied into the DESK.ACCS folder in the SYSTEM folder, where they can be used from any desktop program that follows standard rules for the Apple IIGS.

A new desk accessory does not require a function called main, like standard C programs. Instead, there are four standard functions which must be defined, each of which is called by the Desk Manager to carry out some predefined task. The names of these functions are listed as the first four parameters to the nda command. The next two parameters are integer constants that specify how often the desk accessory is called when it is active (the period parameter), and what kinds of events it handles (the eventmask parameter). The last parameter is the name of the new desk accessory; the name appears in the Apple menu of any desktop program that can call the desk accessory. A more detailed description of each parameter is given below.

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 function 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. The function does not return a value.
action	The action parameter is the name of a function that is called whenever the desk accessory must perform some action. It must declare a single integer parameter, which defines the action that the function should take. 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. The function does not return a value.
init	The init parameter is the name of a function 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 function must define a single integer parameter. The function will be zero for a shut down call, and non-zero for a start up call. The function does not return a value.
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; any other value tells how many 60ths of a second to wait between calls.
eventMask	This value tells the Desk Manager which events your program can handle. The Desk Manager will only call your program with the events you specify in this mask.
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**.

The nda command must be used before the start of the first function in the program. If it is used, you cannot use another nda command or a cda command.

For a complete discussion of new desk accessories, see page 40.

noroot

```
#pragma noroot
```

When ORCA/C creates a program, it generates two object files for each source file it compiles. The first object file has a suffix of .root; this file contains initialization code and a call to main. The second file has a suffix of .a, and contains the various variables and functions declared within the source file.

In a program consisting of multiple source files, the only source file that actually needs a .root segment is the one that actually contains main. This pragma can be used in all of the other source files to tell ORCA/C not to create a .root file, which will make the finished program slightly smaller.

optimize

```
#pragma optimize parm
```

The optimize command is used to control the level of optimization. The various features of the optimizer can make your program smaller and faster, but optimization takes time. During the development cycle, when you are repeatedly compiling and testing your program, the time required by the optimizer is significant enough that you will probably not want to optimize your program

The optimize command requires a single integer parameter. This parameter is actually a series of flags. For each flag, a value of 1 turns the optimization on, and a value of 0 turns it off. Even if new optimizations are added in the future, then, the command

```
   #pragma optimize -1
```

will enable all optimizations, while the command

```
#pragma optimize 0
```

will turn all optimizations off. If you do not use the optimize command, the compiler does not optimize your program.

Bit 0 (a value of 1) controls optimization of the intermediate code. If enabled, the intermediate code is scanned for dead code, which is removed; peephole optimization is performed; and constant expressions are evaluated at compile time.

Setting bit 1 (a value of 2) enables the native code peephole optimizer. This optimization scans the 65816 instructions produced by the compiler, replacing some sequences of instructions with equivalent, but more efficient, instructions.

Setting bit 2 (a value of 4) enables register optimizations. Since the 65816 only has three user registers, all of which have special, preassigned purposes, the compiler does not perform the kinds of register optimizations described in many compiler books. Instead, it scans the 65816 instructions for certain operations that can be avoided. For example, if a

```
ldy #2
```

instruction is detected, but the compiler knows that the value in the y register is already 2, the instruction is removed. While this optimization is technically another form of peephole optimization, the way the checks are performed is substantially different from the other peephole optimizations, so a separate optimizer is used.

Setting bit 3 (a value of 8) turns off the stack repair code normally generated by the compiler. This cuts the size of the code and makes it considerably faster, but if your program has any parameter passing errors, this optimization may cause the program to crash. For help in finding such bugs, see the debug pragma. For a description of legal parameter passing rules, see Chapter 16.

Setting bit 4 (a value of 16) enables common subexpression elimination. This optimization checks for repeated expressions, such as the array calculation for i+1 in

```
a[i+1] = b[i+1] + c[i+1];
```

these common subexpressions are calculated once and saved, generally making the program smaller and faster.

Setting bit 5 (a value of 32) enables loop invariant removal. This optimization checks any code sequence that loops back on itself for expressions which don't change inside the loop, removing these calculations from the body of the loop.

The optimize command is not supported by the small memory version of the compiler. The pragma is accepted, but has no effect on the finished program.

path

```
#pragma path name
```

The path pragma adds a folder to the list of folders the compiler searches when looking for a file.

By default, an include or append statement looks in the current folder (8:) when a file name is enclosed in quote marks, and in the library folder (13:ORCACDefs:) when a file name is inclosed in <> characters. If the compiler does not find the file, it looks in the other of these two folders.

The path pragma adds a new pathname to the list of folders that are searched. The path name itself is the *name* parameter, which is a string. This string can be a partial pathname, in which case it is expanded to a full path name using the normal rules for GS/OS.

Paths added with the path directive are searched immediately after the current prefix, and are searched in the order that they are encountered by the compiler. For an example, assume a program includes these path pragmas:

```
#pragma path "8:myheaders"
#pragma path ":network:projectx:headers:"
```

Then for the include statement

```
#include <stdio.h>
```

the compiler would look for the file using the following paths, stopping as soon as a file was found:

```
13:ORCACDefs:stdio.h
8:stdio.h
8:myheaders:stdio.h
:network:projectx:headers:stdio.h
```

For this include:

```
#include "secrets.h"
```

the compiler would look for the file using the following paths, again stopping as soon as a file was found:

```
8:secrets.h
8:myheaders:secrets.h
:network:projectx:headers:secrets.h
13:ORCACDefs:secrets.h
```

Paths can also be added from the command line using any of the various compile commands. Path names are added with a language-specific compiler option, using the characters `-i` followed by a string containing the search path, as in

```
    run  spy.cc  cc=(-i":network:projectx:headers")
```

Specifying a path name on the command line is completely equivalent to starting the course file with a path pragma.

rtl

```
#pragma rtl
```

C programs normally exit via a Quit call to GS/OS, but some otherwise normal programming environments require a program to exit with an RTL machine language instruction. The rtl command tells ORCA/C to create a program that exits with an RTL rather than a Quit.

The prime example of a program that must exit with a RTL instruction is an initialization program. There are two kinds of initialization programs, permanent initialization programs (file type $B6, PIF) and temporary initialization files (file type $B7, TIF). For information about these kinds of programs, see Apple II File Type Notes for the appropriate file type. For an example of a simple TIF, see the :CC.Samples: disk.

stacksize

```
#pragma stacksize parm
```

All C programs use a run-time stack for calling functions and allocation of locally defined variables. This run-time stack is allocated from bank zero. There is no fool-proof way to determine how much memory will be available in bank zero when your program runs, although programs produced using ORCA/C will issue an error if there is not enough memory. Allocating too little memory for the stack is also a serious problem that will generally result in your program crashing, and can result in corrupting memory being used by the operating system, tools, or other programs. (See the debug command for one way to detect stack overflows.)

All of this means that you should allocate as small a stack as possible, but that there are serious penalties if the stack is too small.

By default, ORCA/C uses the stack allocated by the program launcher; this is 4K for the System 6.0 Finder, PRIZM 2.0 and the ORCA/M 2.0 shell. This memory is used by the startgraph and startdesk functions to allocate direct page space for tools. It is used in all C programs to allocate direct page space for SANE. It is also used for function calls and local variables defined by your program. The stacksize command lets you change the size of the stack. In the desktop environment, you can generally allocate a stack up to about 16K bytes; in the text environment, or in S16 programs, you can often allocate a stack of 32K bytes. The parameter for the stacksize directive is the size of the stack you wish to allocate, in bytes. If it is not a multiple of 256, the number is increased to the next even multiple of 256.

toolparms

```
#pragma toolparms parm
```

There are several instances that arise when using the toolbox where you need to create a function that a tool will call. Tools use a different mechanism for passing parameters than ORCA/C does, so you must tell the compiler to use the toolbox's conventions. Immediately before the function, you should tell the compiler to use the toolbox's mechanism for passing parameters by using the command

```
#pragma toolparms 1
```

Immediately after the function, switch back to the normal calling conventions using the command

```
#pragma toolparms 0
```

You do not have to use the toolparms directive if the function returns void and has no parameters.

You cannot call a function defined this way from within the C program, although the function can call other functions written in C.

This directive changes the stack model used to pass parameters from the one normally used by the ORCA languages to the model used by the Apple IIGS toolbox. It does not change the order of the parameters. In ORCA/C (and most other C compilers, for that matter) parameters are pushed onto the stack starting with the rightmost parameter and working to the left, while in the toolbox (and almost all other high-level languages) the parameters are pushed onto the stack starting with the leftmost parameter and working to the right, just as you would read the parameter list. The toolparms pragma does not affect the order in which the parameters are pushed. To change the order of the parameters, use the pascal qualifier.

See also the databank command.

xcmd

```
#pragma xcmd start
```

The xcmd command is used to create a HyperCard XCMD or XCFN.

Normal C programs must contain a function called main. This function is the one executed when the program starts. XCMDs do not have to have a function called main. Instead, you specify the name of the function that will be called when the XCMD is executed as the *start* parameter to the xcmd command.

The function itself takes a single parameter of type `XCMDPtr` and returns void. When the XCMD is called, HyperCard passes a pointer to a structure containing a variety of information. Information is also returned in this structure.

HyperCard supports a variety of callbacks; these are calls from the XCMD back to HyperCard to perform some action. From ORCA/C, callbacks work almost exactly like tool calls, and from the programmer's standpoint they can be treated as tool calls. The various callbacks are declared in HyperXCMD.h, a header file provided by Apple and included with ORCA/C.

For a definition of `XCMDPtr` and the structure it points to, along with a number of other useful declarations, see HyperXCMD.h. For complete details on how to write HyperCard XCMDs, see the HyperCard folder on the System 6.0 CD ROM..

For a short example of an XCMD, see the :CC.Samples: disk.

Chapter 13 – Basic Data Types

C has a rich variety of data types. This chapter describes those C data types which are built into the language. The next chapter covers derived and user-defined data types.

Some of the information in this chapter deals with the way information is stored internally in the memory of the computer. This information is provided for very advanced programmers who need to write assembly language subroutines that will deal with C 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/C for normal C programming. If it does not make sense to you, or if you will not be using the information, simply ignore it.

Integers

The C language supports four different types of integers, each of which can be signed or unsigned. The four basic integer data types are char, short, int, and long. ANSI C specifies the minimum range for each of these integer data types. While older C compilers may not implement the integers using these same sizes, and new compilers are allowed to make the integers larger, the minimum sizes specified by ANSI C also happen to the the most common size for integers on microcomputers, and are the sizes used by ORCA/C.

Integers defined as char are the smallest. Char values require one byte of storage. Signed char values can range from -127 to 127; unsigned char values have a range of 0 to 255. The C language allows the compiler implementor to decide if char values will be signed or unsigned by default; in ORCA/C, char values are unsigned by default. All other integer types are required to be signed integers by default.

Short and int variables are the same size. They require two bytes of storage. Signed values can range from -32767 to 32767, while unsigned values can range from 0 to 65535.

Long integers require four bytes of storage. Signed long integers range from -2147483647 to 2147483647, while unsigned long integers range from 0 to 4294967295.

Internally, all unsigned integers are represented as binary values. Signed integers are represented as two's complement numbers. All of the integers that occupy more than one byte of storage are stored with the least significant byte first, proceeding to the most significant byte.

Reals

ORCA/C supports four storage formats for real numbers. In all cases, calculations are performed using SANE or the Innovative Systems' 68881 floating point card using ten byte intermediate values. With some of the formats, the numbers are stored with less precision, so that you may see different results if you compare one equation that does not save intermediate results with a mathematically equivalent set of equations that store intermediate values in the less precise number formats.

Float and double are required in all C compilers. Float numbers are stored in the IEEE floating point number format, with the least significant byte first. They require four bytes of storage. Float values are accurate to seven decimal digits, and allow exponents with a range of about 1e-38 to 1e38.

Double numbers require eight bytes of storage each. They are stored least significant byte first, using the IEEE floating point format. They are accurate to fifteen decimal digits, and allow exponent ranges from 1e-308 to 1e308.

Extended numbers are peculiar to implementations of C on Apple computers. They require ten bytes of storage each. The storage format is an extended version of the IEEE format, with the values stored least significant byte first. They are accurate to nineteen decimal digits, with an exponent range of 1e-4932 to 1e4932. This is the format used internally by both SANE and the 68881 floating point coprocessor, so there is no loss of precision if intermediate variables are of type extended. In addition, floating point calculations are performed faster if you are using SANE and all numbers are stored in extended format, since a conversion from the original format to extended format is always a first step in doing a calculation with SANE. Exactly the opposite is true if you are using the 68881 floating point card. In that case, conversions are done in hardware. Float variables are faster with the 68881 card, since it takes less time to pass the number to the card and retrieve the result with the shorter formats.

Comp format numbers are signed integers that require eight bytes of storage. They range from -(263-1) to 263-1. While comp numbers are integers, they are treated as a special form of floating-point number.

Pointers

Pointers are represented internally as four-byte unsigned numbers, with the least significant byte stored first. A value of zero is used to represent a null pointer. Using type casting, pointers can be treated as unsigned long integers in mathematical equations with no loss of precision.

Chapter 14 – C Programs

The Anatomy of a C Program

C programs are made up of a series of declarations. These declarations consist of declarations of global variables and declarations of functions. In most C programs, some of the functions and global variables are defined in libraries or separately compiled modules. The compiler is informed of their existence through interface files.

While macro preprocessor commands are used in C programs to define constants and implement some operations, the macro preprocessor is not technically a part of the C language. For a description of the macro preprocessor, see Chapter 12.

The Function main

All C programs except classic desk accessories and new desk accessories must have a single function called main. This function is the first one called; when you return from main, program execution stops.

It is not important where main appears in a program; if the program consists of several modules, main can appear in any of them, regardless of the order in which the modules are compiled or linked. The important point is that a function named main must appear in exactly one of the modules.

You may declare the function main as either a function returning an integer or as a function returning void. Declaring it as a function returning an integer allows you to send a return code back to the caller. If your program is designed to run from script files, this is an important step. It allows you to tell the script file that the program terminated normally by returning zero, or to inform the shell that an error occurred by returning a non-zero value. If you do not return a value from main, the value reported to the shell is unpredictable.

Argc and Argv

The shell passes two arguments to the function main. These arguments can be ignored or used; to ignore them, don't code any parameters to the function. These arguments allow you to look at the command line used to execute your program. The sample program below shows how these arguments are used.

```
/* echo the command line arguments */

#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];

{
int i;

for (i = 0; i < argc; ++i)
   printf("%2d: %s\n", i, argv[i]);
}
```

In this program, you see argc and argv defined as parameters to main. While you do not have to define argc and argv, if you do, they must be defined in the order shown. When your program executes, the command line used to execute your program is scanned. If you executed the program from the text environment or from the shell window on ORCA/Desktop, the command line is the line you typed or that appeared in a script file that actually executed your program. If you execute your program using the RUN command from the shell, or using the Compile to Memory or Compile to Disk commands form the desktop, the command line will be empty.

After I/O redirection characters are removed, aliases are expanded, and any shell variables are expanded, the command line is broken into strings. These strings are formed by scanning the command line from left to right. A string consists of any sequence of characters that contain no white space. Argc is the number of strings found, while argv is an array of pointers to the strings. If any strings are present, the first (argv[0]) is always the name of your program.

To see how these strings are built, and to experiment with the way lines are scanned, type in the sample shown above. If you are using the desktop environment, be sure to turn debugging off. Then execute the program from the text shell or shell window, supplying a variety of parameters.

Separate Compilation

C programs can be divided into more than one module, and each module can be compiled separately from the others. The resulting object files can then be linked together, producing a single program.

The object files that make up a program are not limited to object files created by the C compiler. You can mix C code with code written using ORCA/Pascal, the APW assembler or the ORCA/M assembler.

For a description of the mechanics of linking programs together, see Chapter 4. Chapter 15 covers the storage types used to hide variables and functions from other modules, and the storage types used to make functions and variables from one module available to other modules.

Interface Files

Interface files are used to tell the C compiler about functions and data that appear in libraries or separately compiled modules, and to define macros that are used in more than one module. The distinction between an interface file and a file that contains C source code is merely a matter of convention. Interface files are created using the same text editor that you use to create programs.

They can be modified by you. You can even extract declarations from an interface file and embed them directly in your program.

In general, interface files come from two sources. ORCA/C comes with a large number of interface files that help you use the standard C libraries and the Apple IIGS toolbox. These interface files are located in the ORCACDefs folder of your library directory. To include one of these interface files in your program, use a #include command, and place the name of the interface file in brackets, like this:

```
#include <stdio.h>
```

When you read the descriptions of the standard C libraries in Chapter 19, each description shows the #include command that should be in your program to call the function. With some functions, you can avoid using the interface file, and that was quite common in older C compilers. ORCA/C implements function prototypes, though, which allow the compiler to perform sophisticated checking on the parameters you pass to a function. If you do not use the interface file, the compiler cannot make these checks. Since illegal parameters account for a large number of bugs is C programs, many of which can crash the executing program, it is a good idea to make use of the include files.

The second major source of include files are those that you write yourself. These can be new libraries that you have created, in which case the interface file should be placed in the ORCACDefs folder and accessed like any other library, or they could be interface files used in a program that is made up of separately compiled modules, in which case you would use quote marks around the file name in the #include command, instead of brackets. For example, if you need to include an interface file called commondefs in your program, and the interface file is in your current working directory, you could use the statement

```
#include "commondefs"
```

Chapter 15 – Declarations

Declarations

A program is made up of one or more declarations, one of which must be a function called main. The basic syntax for a program is shown below.

```
program:   {initialized-declaration | function-definition |
     asm-function-declaration}*
initialized-declaration:
     declaration-specifiers initialized-declarator
     {',' initialized-declarator}* ';'
initialized-declarator:  declarator {'=' initializer}
function-definition:
     {declaration-specifiers} declarator {declaration-list}
     compound-statement
declaration-list:  declaration {',' declaration}
declaration:  declaration-specifiers declarator-list ';'
declaration-specifiers:  {storage-class-specifier | type-specifier}*
declarator:
     identifier |
     '(' declarator ')' |
     function-declarator |
     array-declarator |
     pointer-declarator
```

Storage Classes

```
storage-class-specifier:  [auto | extern | register | static |
     typedef]
```

Each function or variable has a storage class. This storage class may be assigned explicitly by starting the declaration with one of the storage class names shown above.

For global variables and functions, if no storage class is specified, a storage class of extern is assumed. There is a difference, however, between a variable or function defined with the storage class extern, and a variable or function defined with no explicit storage class. When the storage class of extern is specified explicitly, the compiler creates a reference to the variable or function, but it is not defined. Instead, the compiler expects that the variable or function will be resolved later by the linker. When no storage class is specified, the compiler generates a variable or function that can be accessed from outside of the current module. Only one such occurrence is allowed in any one program.

```
extern int i;          /* an integer declared in another module */
extern foo();          /* a function declared in another module */

int i;                 /* an integer which can be accessed from */
                       /* another module                       */
```

For variables defined within a function, a storage class of auto is assumed. Auto variables are available throughout the function, but cannot be accessed from outside of the function. Auto variables defined within a function lose their value when the function returns to the caller.

```
int i;                 /* auto integer */
```

The static storage class has three different meanings, depending on where it is used. Functions definitions of type static are not exported to the linker. They become private to the current module, and cannot be accessed from other modules, whether or not the code is turned into a library. Function declarations (where a function is declared, but the statements that make up the function are not specified) of type static indicate that the function body will appear later in the current source file, and that the function will be of storage class static. Global variables defined with a storage type of static cannot be accessed from outside the current module. Local variables defined with a storage type of static remain intact after leaving the function. That is, any value assigned to the variable during a function call is still available when the function is called the next time.

```
static int i;          /* static integer */
static foo();          /* a function that will be defined later */
```

The register storage class can only be used with variables defined within a function. It has the same meaning as auto, but instructs the compiler to handle the value in the most efficient way possible, placing the value in a register if one is available. The 65816 does not have enough user registers to leave values in a register, so in ORCA/C, this storage class has exactly the same meaning as auto.

Unlike any other storage class, this storage class can be used with a function parameter. Again, though, ORCA/C does not do anything special with the parameter if the storage class is register.

```
register int i;        /* register int */
```

The storage class typedef is used to define types. The name of the type appears where the variable name would have appeared in any other declaration, and the new type has the same type that the variable would have been assigned if typedef were not used. The new type can then be used as a type specifier (see the next section).

```
typedef int *intPtr;   /* definition of a type: pointer to integer */
intPtr ip;             /* defining ip as a pointer to an integer */
```

Type Specifiers

```
type-specifier:
     {volatile} {const}
     [enumeration-type-specifier |
     floating-point-type-specifier |
     integer-type-specifier |
     structure-type-specifier |
     identifier |
     union-type-specifier |
     void]
     {volatile} {const}
floating-point-type-specifier:
     float |
     {long} double |
     extended |
     comp
integer-type-specifier:
     signed |
     {signed} int |
     {signed | unsigned} short {int} |
     {signed | unsigned} long {int} |
     unsigned {int} |
     {signed | unsigned} char
```

The type specifier specifies the type for a variable or function. The type specifier for a function is optional, whether or not a storage class specifier is used; if omitted, int is assumed. A variable declaration must include either a storage class specifier or a type specifier (and can, of course, include both). If the type specifier is omitted, a type specifier of int is assumed.

Type specifiers for enumerations, structures and unions are covered in separate sections, below.

Floating-point and integer type specifiers are used to declare variables, pointers, and arrays of the various built-in data types. They all have a similar structure. Floating-point numbers come in three sizes, as detailed in Chapter 13. The smallest size requires four bytes of storage; this size is designated by the type specifier float. Double-precision numbers, requiring eight bytes of storage, can be specified as double or long double. SANE extended format numbers, requiring ten bytes of storage each, have a type specifier of extended.

SANE also supports an eight-byte signed integer format called comp. Comp variables are treated as a special case of floating-point numbers, despite the fact that they do not support exponents or non-integer values.

```
float a,b;              /* two floating-point variables */
double sum;             /* sum is double-precision */
long double sum2;       /* sum2 is double-precision */
double *dp;             /* pointer to a double precision variable */
extended a[10];         /* array of 10 extended variables */
extern float sum();     /* external function returning float */
```

```
    comp big;              /* eight-byte signed integer */
```

Integer type specifiers have a variety of formats, but are easy to understand once you know the systematic way they are defined. There are basically four types of integers: char, short, int and long. Since short and long are considered to be modifications of the basic type of int, you can follow either of these type specifiers with int, but you are not required to do so, and most programmers do not. Any of the four basic integer data types can be signed or unsigned. By default, they are all signed. If you like, you may prefix any of the types with signed, ensuring that the value is signed even if the source code is ported to another compiler. This is a good idea for char variables, since some compilers use signed char by default, but doesn't matter on the other integer data types, which are always signed by default. You may also create unsigned versions of each of the data types by prefixing it with unsigned. When an unsigned integer suits your needs, it is a good idea to use them. On the Apple IIGS, code for comparing unsigned values is much more efficient than code for comparing signed values. The examples below show some typical cases.

```
    foo();                 /* function returning int */
    int i,j,k;             /* three signed integer variables */
    signed char *cp;       /* a pointer to a signed character */
    unsigned long l[10];   /* an array of 10 unsigned long integers */
    long int l;            /* a long int */
    short sum;             /* a short integer */
```

When you define a type using the typedef storage class, the name of the type (and identifier) can be used as a type specifier in later declarations. For example, let's assume that a pointer to an integer has been defined as a new type:

```
    typedef int *intPtr;
```

You can now use this type to define variables and function results.

```
    intPtr ip;       /* ip is a pointer to int */
    intPtr search(); /* search is a function returning a pointer to int */
```

Void is a special type specifier that declares a function that does not return a value, or that a pointer points to a generic type. There are three places where this type specifier can be used. Functions returning void are functions that do not return a value. In other languages, these are known as subroutines or procedures, and the term function is only used when a result is returned. Older C programs, written before void became a part of the language, defined functions returning int for this purpose, depending on C to dispose of the unneeded value. There are two reasons to use void explicitly on new programs: it avoids confusion, and possibly errors, and ORCA/C generates more efficient code when exiting a function of type void than when exiting a function returning int.

The second place where void is used is to define an untyped pointer. These should be avoided when possible, but are occasionally useful. The most common place to use this type of pointer is in functions that require a pointer to a data structure whose type can change from call to call.

Finally, void is used in type casting to cast a pointer to an untyped pointer, making the pointer type-compatible with any other pointer. In older C programs, a pointer to char was generally used for pointers to an unspecified type.

```
void *untyped_pointer;  /* an untyped pointer */
void fn(void);          /* a function that returns nothing */
```

Volatile is a type specifier that modifies a type. If it appears with no other type, a type of int is assumed. Volatile can appear before or after any other type specifier. Volatile is an instruction to the compiler, telling it that the value can be changed in ways that are not under control of the compiler. Examples would be hardware ports like the Apple IIGS keyboard latch at 0x00C000, or variables that will be changed by interrupt handlers. The compiler is not allowed to do optimizations on volatile variables that would delay references to the variable, or change the order in which references occur.

Const is another type specifier which modifies an existing type. If it appears with no other type, a type of int is assumed. Const can appear before or after any other type specifier. Const variables cannot be changed by the compiler, although they can be initialized. Any attempt to change the variable will result in a compile-time error.

Enumerations

```
enumeration-type-specifier:
     enum {identifier} |
     '{' enumeration-constant {',' enumeration-constant}* '}'
enumeration-constant:  identifier {'=' expression}
```

Enumerations provide an easy way to create integer constants. They are used in situations where a series of names make the program easier to read than using integer values. For example, when dealing with the 640 pixel wide screen, you have four colors available. While they can be changed, these colors default to black, purple, green and white, with integer values for the colors of 0 through 3. You could write a program that uses the integers 0 through 3 to represent these colors, or you could use preprocessor macros to create names for the colors. Enumerations provide yet another alternative. The enumeration

```
enum {black, purple, green, white} pencolor;
```

defines the integer variable pencolor, and simultaneously defines four integer constants. The values of these constants start at zero and increment by one, so that

```
printf("%d\n", green);
```

would write 2 to standard out.

An enumeration can be used to define a new enumeration type by including an enumeration tag before the list of enumeration constants. For example,

```
enum color {black, purple, green, white} pencolor;
```

In this case, we have still defined the variable pencolor, but you can define an enumeration type without creating variables. The enumeration type color can now be used to define other variable with the same enumeration type.

```
enum color frameColor, buttonColor, menuColor;
```

By default, enumeration constants are assigned values by setting the first to zero, then incrementing each successive enumeration constant by one. It is possible, however, to set each constant to an explicit integer value. Once this is done, if the next constant does not have an explicit value, a value is assigned to it by incrementing the last constant value by one, as before. To do this, follow the constant by an equal sign and a constant expression, as shown below.

```
enum month {January = 1, February, March, April, May, June, July,
   August, September, October, November, December};
enum ages {George = 28, Matilda = 28, Jimmy = 5, Cindy};
```

In the first case, integer values for months are assigned sequentially, but a starting value of one was needed for January. The second example shows the ages of everyone in a family. Note that it is legal to specify the same value for two different enumeration constants. Cindy's age would be six, since no specific value was specified, and the previous constant has a value of five. While these examples do not show it explicitly, any integer value may be used, including negative values.

Arrays

```
array-declarator:  declarator '[' {constant-expression} ']'
```

Arrays are indexed lists of data, where all of the data in the array has the same type. In C, you can form arrays of any type except functions (although arrays of pointers to functions are legal), including arrays of structures, unions, or other arrays. To define an array, follow the declarator for the variable with a left bracket, a constant expression, and a right bracket. The constant expression is evaluated, and used to determine how many elements are in the array. The first element is indexed with a value of zero, and remaining indices are formed by adding one to the previous index.

For example, to form an array of ten integers, and then fill the array with the numbers one through ten, you could use the following statements:

```
int a[10],i;

for (i = 0; i < 10; ++i)
   a[i] = i+1;
```

It is important to keep in mind that the lowest index is zero, and the highest is one smaller than the number of subscripts. So, in the example just given, there is no tenth element of the array. That is, a[10] does not exist, and storing a value at a[10] may cause the program to crash.

Arrays of more complicated elements are formed the same way as arrays of simple variables. For example, to form an array of points, where a point is defined as a structure containing three float numbers, you could use

```
struct point {float x,y,z;};
struct point list[100];
```

C does not allow multiply subscripted arrays, but it does allow arrays of arrays, which amount to the same thing. For example, to create a ten by ten matrix of float values, you would use the declaration

```
float a[10][10];
```

The following code shows how to access elements of the array by forming the identity matrix (a matrix with ones along the diagonal, and zeros everywhere else).

```
for (i = 0; i < 10; ++i) {
   for (j = 0; j < 10; ++j)
      a[i][j] = 0.0;
   a[i][i] = 1.0;
   }
```

In some cases, it is permissible to define an array without specifying the size. This is true for singly-dimensioned arrays that are external to the current unit, arrays that are passed as a parameter to a function, and arrays that are initialized. In the first two cases, the compiler does not need to know the exact size of the array, since storage for the array is allocated elsewhere. It is up to you to make sure that you do not access values beyond the end of the array. Even in these cases, if the array is an array of arrays, all subscripts but the first must be specified. In the case of an initialized array, the compiler determines the size of the array by counting the initializers. The following example shows legal and illegal declarations of external arrays.

```
extern int a[][10];     /* legal:  ? by 10 array of int */
extern int b[10][];     /* NOT LEGAL! */
int a[] = {1,2,3};      /* legal:  three elements */
```

The array is stored in memory in such a way that the first subscript indexes the slowest. To visit sequential locations in memory, you would use a[0][0], a[0][1], a[0][2], and so forth.

The size of an array is the number of elements in the array times the size of an array element. In the previous example, each floating-point number requires four bytes of storage, so an array with ten elements requires forty bytes of storage. The array of these ten element arrays also has 10 elements, so the array a requires 400 bytes of storage. In a program, you would normally use the sizeof operator to determine the size of an array.

Using the small memory model (which is the default), the largest single array that is allowed is 64K bytes. In addition, the total length of all variables, libraries, and code from your program that is not placed in another segment using the segment command is 64K bytes. The large memory model lifts these restrictions at a price of less efficient code. With the large memory model, an array can be as large as the largest free block of memory, so long as there is enough memory to load all global variables and static segments. With a memory card with enough memory, you could manipulate an array that is nearly eight megabytes long.

Pointers

```
pointer-declarator:  '*' {type-specifier}* declarator
```

A pointer is a data type that holds the address of another object. Pointers are represented internally as unsigned four-byte values. The value is the address of the object pointed to by the pointer. A pointer is defined by placing an * to the left of the variable, as in

```
int *ip   /* ip is a pointer to an integer */
```

Two operators are used to handle pointer types. The * operator, when used before a pointer, tells the compiler to load the object pointed to by the pointer. The & operator, when used before any l-value, tells the compiler to load the address of the object, rather than the actual object. The following code fragment uses the & operator to cause ip to point to the integer i, then sets i to four using a reference through the pointer, and finally prints the value of i directly and as a reference through the pointer ip.

```
ip = &i;                      /* ip now points to i */
*ip = 4;                      /* i now has a value of 4 */
printf("%d = %d\n", i, *ip);  /* both values printed will be 4 */
```

Pointers can be defined to point to any other data type except a bit field, including other pointers, functions or void.

ANSI C allows a pointer to have a type specifier before the declarator. This is generally used with the type specifiers volatile and const, but the type specifier is not restricted to just those two types. Const and volatile can be used in conjunction with other types, so they modify the existing type of the pointer. For example, the declaration

```
int * const ip;
```

declares a pointer to an integer, and states that the pointer cannot be modified. If a type specifier other than const or volatile is used, it overrides the existing type. For example, the following declaration creates a pointer to a float variable.

```
int * float fp;
```

The best that can be said for such a declaration is that it is legal. Such declarations are confusing, and should not be used in real programs.

There are many other operators that can be used with pointers in C, including the array subscript operator, the increment and decrement operators, addition and subtraction, and comparisons. Chapter 17 discusses these topics.

Structures

```
structure-type-specifier:
     struct identifier |
     struct {identifier} '{' field-list '}'
field-list:  component-declaration {',' component-declaration}*
component-declaration:  type-specifier component {',' component}*
component:
     declarator |
     {declarator} ':' expression
```

Structures are collections of variables that do not have to have the same type. With the exception of bit fields, each variable in a structure has a name, just as variables in other parts of the program do. (The name is optional on a bit field.) Variables in a structure are called components, and the names of the variables are called component names. Each of the component names must be distinct from any other component name in the same structure. A structure can contain elements of any type except a function or void. Pointers to functions are allowed, but a function declaration cannot appear as a part of a structure.

It is possible to define a structure type without defining variables, and then define variables later; to define a structure variable without defining a type; and to define both a type and variables at the same time. We will look at the general form by examining a structure that implements a linked list of points, with each point consisting of three float numbers.

```
struct point {struct point *next; float x,y,z;} p, *list;
```

This structure contains four elements: next, x, y and z. X, y and z are the three float variables that record the position of the point. Next is a pointer to another structure of type point. Note that it is permissible to use the definition of the structure point, so long as we are defining a pointer to the structure. Trying to recursively define the structure would be illegal, since the compiler could not compute the size of the structure. Finally, two variables are defined. The first, p, is a structure of type point, while the second is a pointer to a structure of type point.

Further variables of type point may be defined by using the structure name (called the tag) without redefining the form of the structure. For example, we could define three more points like this:

```
struct point p1, p2, p3;
```

The tag field of the structure is optional. If you omit the tag field, however, no other variables of the same type can be created later in the program.

The size of a structure can be determined by summing the size of each component. In the structure point, there are four elements, each of which is four bytes long, so the size of the variable p is sixteen bytes. The values appear in memory in the same order in which they are defined. In our example, next occurs first, followed by x, then y, and finally by z.

To access a component of a structure, the name of the structure is followed by a period and the name of the component name of the variable to be accessed. For example, to initialize the point p to (1.0,2.0,3.0), with a null pointer, we would use the assignments

```
p.x = 1.0;
p.y = 2.0;
p.z = 3.0;
p.next = NULL;
```

To set p1 to the reflection of p through the origin, we could use the statements

```
p1.x = -p.x;
p1.y = -p.y;
p1.z = -p.z;
```

To access an element of a structure through a pointer to the structure, the -> operator is used. For example, to move the point pointed to by list two times further away from the origin, you would need to multiply each of the coordinates in the point by two. The following statements will make this change.

```
list->x = list->x*2.0;
list->y = list->y*2.0;
list->z = list->z*2.0;
```

Structures may be assigned to other structures of the same type. For example, using the declarations from our examples, the following assignments are legal.

```
p1 = p2;
*list = p;
p3 = *list;
p1 = *p.next;
```

Functions can be defined which accept structures as parameters and which return structures as results. The following example shows a function which takes two points as parameters, and returns a point midway between the two inputs.

```
struct point midpoint(struct point a, struct point b)

{
struct point c;
c.x = (a.x + b.x) / 2.0;
c.y = (a.y + b.y) / 2.0;
c.z = (a.z + b.z) / 2.0;
return c;
}
```

Using this function to find a point midway between p1 and p2, storing the result in p3, we would code

```
p3 = midpoint(p1,p2);
```

Bit fields are a data type peculiar to structures. A bit field is an integer data type which is bit-aligned, rather than byte-aligned. They can be used to store integers in a very compact fashion, or to access bits within a byte of memory. To see how this is done, we will define a simple structure using bit fields.

```
struct pack {
   unsigned field1 : 9;
   unsigned        : 2;
   unsigned field2 : 4;
   } v;
```

Each of the fields in this structure requires a specific number of bits, not bytes. Field1 requires nine bits; it takes up all of the first byte, and the first (most significant) bit of the next byte. It is an unsigned integer, with a range of 0 to 511. The next bit field has no name; it is only there to reserve a specific amount of space. Field2 requires four bits of space, and has a range of 0 to 31. The variable v, then, requires fifteen bits of space. In all cases where a series of bit fields does not end on a byte boundary, the compiler in effect creates another field to fill out the bits to an even byte boundary. In this example, one bit must be added, so that the variable v uses sixteen bits (two bytes). The two bits between field1 and field2, and the bit that comes after field2 cannot be accessed from the program, and their values are not predictable.

Bit fields may be interspersed with other variables in a structure, as shown in the second example, below.

```
struct data {
   unsigned color640 : 2;
   int i;
   unsigned color320 : 4;
   } v;
```

The variables which are not bit fields must start on a byte boundary. Thus, the variable color640 only requires two bits, but since the variable i must start on a byte boundary, the entire structure requires four bytes of storage (one each for color640 and color320, and two for i). A simple rearrangement of the structure reduces the memory requirements by one byte.

```
struct data {
   unsigned color640 : 2;
   unsigned color320 : 4;
   int i;
   } v;
```

ORCA/C supports both signed and unsigned bit fields. Unsigned bit fields are stored in binary format. The range that can be represented is 0 to 2b-1, when b is the number of bits in the bit field. Signed values are represented in two's complement form, giving a valid range of -2b-1 to 2b-1-1. The maximum size for a bit field is thirty-two bits.

Accessing bit fields is very inefficient compared to accessing integers. If speed is an issue, avoid their use.

Programs that make use of bit fields are difficult to port from machine to machine, so bit fields should be used sparingly, if at all. Not all compilers support signed bit fields. The maximum size for a bit field varies from compiler to compiler, and the way bit fields are stored internally also varies.

Unions

```
union-type-specifier:
     union identifier |
     union {identifier} '{' field-list '}'
```

A union has a syntax similar to that of a structure and, like structures, contains named elements called components. Unlike structures, bit fields are not allowed in unions. Like structures, components of unions can be of any type except function returning or void. Within any one union type, each component name must be unique. No component can be of the same type as the union in which it appears, but components can point to unions of the same type.

Components of unions are accessed the same way as components of a structure. Unions can be assigned to other unions of the same type and returned as the result of a function, just like structures. The major difference between structures and unions is in how storage is allocated for the components. In a structure, memory is assigned to each variable in turn, and the total size of the structure is the sum of the sizes of all of the components. In a union, each of the variables overlaps. The size of the union is the size of the largest component, since only one of the components is stored in the union at any one time. Unions are most useful in situations where two or more types of data will be stored in a location, but the two do not need to be stored at the same time. As an example, let's consider a program that evaluates expressions, and needs to store variable values. We will assume that the name of the variable must be stored, and that it is limited to ten characters. We will also assume that the variables can be integer or float. Since a single variable couldn't be both integer and float at the same time, we will use a union to overlay the int and float variables, saving space.

```
enum kind {integer, real};
struct variable {
   char name[11];
   enum kind vkind;
   union {
      int ival;
      float rval;
      } val;
   };
```

This example also shows the use of a tag field to record the type of value stored in the union. While this is not required, it is often useful. Without a tag field, your program must have some other way of figuring out if the value stored in the union is an integer or a floating-point number.

To evaluate the storage requirements for the structure, we first determine the size of the union. Ival is two bytes long, while rval is four bytes long. The union, then, is four bytes long. The name of the variable requires eleven bytes, and the tag field, which is an integer, requires another two bytes. The total size of the structure, then, is seventeen bytes.

Initializers

```
initializer:
     expression |
     '{' initializer-list {','} '}'
initializer-list:  initializer {',' initializer}*
```

When a variable is defined, the declaration can include an initializer which specifies the initial value for the variable. Variables with a storage class of static and extern can only be initialized with a constant expression. All static and extern variables that are not explicitly initialized are initialized with a value of zero. Function parameters cannot be initialized.

Integer, enumeration and floating-point variables are initialized by following the variable name with an equal sign and the initial value. The initial value can be inclosed in braces, but the usual practice is to omit the braces. Non-constant expressions can be used to initialize variables that have a storage class other than extern or auto. In that case, the compiler generates the same code that would be generated if the variable was initialized via an assignment statement.

The following examples show some legal initializations.

```
i = 4;
auto j = i*4;
static float x = 1.0, y = 2.0, z = 0.0;
```

An enumeration can be initialized to an enumeration constant or to an integer value. As with integers, the expression must be a constant expression if the storage class of the enumeration variable is extern or static. For example, we can define an enumeration, declare a variable, and assign the variable an initial value all in one step, like this:

```
enum color {black,purple,green,white} pencolor = white;
```

Initialization of pointers follow the same rules as initialization of integers. The following operands are all constants, and can be used in a constant initializer for a pointer:

1. The integer constant 0 (or the preprocessor macro NULL).
2. The name of a static or external function.
3. The name of a static or external array.
4. The & operator when applied to a static or external variable.
5. The & operator when applied to a static or external array with constant subscripts.

6. A non-zero integer constant cast as a pointer type.
7. A string constant.
8. An integer constant added or subtracted to any of the items 3 through 7.

Arrays are initialized by enclosing the initializers in braces, and separating them with commas. Each initializer in an array must be constant expressions. The example below shows the initialization of a ten-element array.

```
int a[10] = {1,2,3,4,5,6,7,8,9,10};
```

Multi-dimensioned arrays follow the same pattern:

```
int a[2][2] = {{1,2},{3,4}};
```

In this case the initial values are as follows:

a[0][0]	1
a[0][1]	2
a[1][0]	3
a[1][1]	4

There are four special rules used when dealing with array initializers. First, if an initializer for an array contains fewer values than the size of the array, the remainder of the array elements are initialized to zero. The second rule is that the size of an array does not need to be specified. In that case, the size of the array is derived from the number of initializers. For example, the following array has five elements, initialized to one through five.

```
int a[] = {1,2,3,4,5};
```

Another special case is an array of characters. An array of characters can be initialized using a string, as in

```
char str[15] = "Hello, world.\n";
```

The compiler automatically places a terminating null character at the end of the string constant. Finally, if the initializer list contains the proper number of elements, all embedded braces may be omitted. Repeating our previous example using this rule, we have

```
int a[2][2] = {1,2,3,4};
```

Structures are initialized by enclosing all of the values in braces and separating them with commas. As with arrays, if the number of values supplied is less that the number of variables in the structure, the remaining elements are set to zero. In addition, if a structure is embedded in another structure, and the correct number of initializers are supplied, the braces around the initializers for the embedded structure may be dropped.

```
struct point {float x,y,z;} p1 = {1.0,2.0,3.0};
struct line {point p1,p2} line1 = {1.0,2.0,3.0, 2.0,3.0,4.0};
struct line2 = {{1.0,2.0}, {2.0,3.0}};
```

The first component of a union can be initialized with an expression resulting in the same type as the component. If the storage class for the union is auto, the first component must have a type that could be initialized if the component were specified as a separate auto variable. For example,

```
union nums (float f; int i;} x = 0.0;
```

For all global or static variables, if no initializer is given, the variable is initialized to zero.

The way initializations are handled by the compiler depends on where the variable is declared. Global and static variables are initialized by setting the memory area reserved for the variable to the correct initial value. Auto and register variables defined in a function are initialized using code that is equivalent to an assignment statement.

If a variable is declared with an explicit storage class specifier of extern, it cannot be initialized.

Constructing Complex Data Types

The facilities described in this chapter show how to declare variables in a variety of simple data types. You can also combine many of these attributes, creating complex definitions in a single step. For example,

```
float (*arr[10])();
```

defines a ten-element array of pointers to functions returning float. With some restrictions, any combination of storage class specifiers, type specifiers and declarators can be used to form a type. These restrictions are:

1. Void can only be used as the type returned by a function or pointed to by a pointer.

2. Arrays, structures and unions may contain pointers to functions, but may not contain functions.

3. Functions can return structures, unions or pointers to arrays, but they cannot return an array.

4. Functions cannot return another function. They can return a pointer to another function, however.

To read the type of a variable, it is important to understand the precedence of the declarator operators. Basically, declarators that appear to the right of the variable (arrays and functions) have a higher precedence than declarators appearing to the left (pointers). The declarator that is closest to the variable has the highest precedence. Parentheses can be used to override the normal precedence. To see how this is done, we will use an absurdly complex declaration.

```
struct point {float x,y,z;} *(*(*(*x)())[10])();
```

Starting at the variable, we can read off the type. The variable x is a pointer to a function returning a pointer to a ten-element array of pointers to functions returning pointers to structures that contain three float variables. In a real program, such a declaration would probably not be needed. If such a declaration was needed, it would be easier to define using typedefs.

Scope and Visibility

All declarations that appear outside of a function are available from the declaration point to the end of the source file, unless some other declaration masks the declaration.

Function parameters are available throughout the function unless some other declaration masks the parameter. Parameters cannot be accessed from outside the function, even from a function called by the one that defined the parameters. (Pointers to variables can, of course, be passed to another function, and the value changed, but the called function cannot access the variable using the original name.) A function parameter can have the same name as a global variable or function, in which case the global variable or function cannot be referenced from the function. For example, the code fragment

```
int count;

void test(float count)

{
/* statements go here */
}
```

is legal. Any function declared after the global declaration of count can use or modify count, but that variable cannot be used or modified from within the function test. Inside test, count refers to the float parameter.

Variables defined at the top of the function body are also available anywhere in the function, and are not available from outside of the function. Variables defined at the top of the function body cannot duplicate the names of parameters, but they can duplicate global variables or functions.

```
int count;

void test(void)

{
double count;
...
}
```

Within the function, references to count use the double variable.

Any compound statement within a function can declare variables whose scope is limited to the duration of the compound statement. These variables cannot be accessed from outside the compound statement, although they can be used from compound statements embedded in the one where the variable is declared. Variables defined within a compound statement can reuse the names of global variables or functions, parameters, or variables defined in the program body or other compound statements. For example, the following function will print 1, 2 and 1 to standard out.

```
void print(void)

{
int i;

i = 1;
printf("%d\n", i);
   {
   int i;

   i = 2;
   printf("%d\n", i);
   }
printf("%d\n", i);
}
```

Preprocessor macros are available from the point they are defined to the end of the source file or to the undef command that removes the macro definition. Note that since macro expansion conceptually occurs before the program is compiled, once a macro is defined, and occurrence of the macro name will be replaced by the macro body, even if that occurrence appears to be defining a new variable. For example,

```
#define name sally
int name;
```

defines a new variable called sally, not a variable called name.

Labels in a function are available throughout the function in which the label appears.

There are several cases in C where a name can be duplicated within the same function, or in the global declaration area. The ability to do this is called overloading. The reason is that C maintains several different tables of symbols. The overloading classes are:

1. Preprocessor macro names
2. Statement labels
3. Structure, union and enumeration tags
4. Components of structures or unions
5. Other names (variables, functions, typedef names, enumeration constants)

What this means is that the same name can be reused within a function, so long as it is not used more than once within the same overloading class. Another important point is that each structure and union has its own overloading class; thus, it is legal to define two structures or unions, and to use the same component names in the various structures and unions. This is, in fact, one of the few places where it is reasonable to take advantage of the overloading classes. For example, if you are defining more than one kind of linked list, it would be very reasonable to name the pointer to the next structure next in each of the structures. The following example shows each of these rules in effect.

```
void test(void)

{
#define name gorp
int name;
enum name {Fred, Joe};
struct s {int name};
union u {int name};
goto name;
name: ;
}
```

Chapter 16 – Functions

Declaring a Function

```
function-declarator:
     declarator
     '('
     [parameter-type-list {',' '.' '.' '.'} |
     {parameter-list} ]
     ')'
     {inline '(' long-integer ',' long-integer ')'}
asm-function-declaration: asm identifier '{' {asm-line}* '}'
```

C function declarations can be intermixed with other declarations throughout the program. The only restriction is that a C function cannot be defined within the body of another function.

A function declaration looks very much like the declaration of any variable. The two major differences are than functions do not have to have a type specifier, and the type is always function returning something. When a function is declared without a type specifier, a type specifier of int is assumed. For example, the following two declarations are almost identical, but the first defines a function returning a pointer to int, while the second defines a pointer to a function returning int.

```
int *f();
int (*f)();
```

It is also possible to declare a function automatically. This happens when a function that has not been declared is used in an expression, as in

```
x = test();
```

When a function is declared automatically, it is assumed to be a function returning int. Any later declarations or definitions must also be for a function returning int. To avoid problems, and to make effective use of function prototypes, it is a good idea to declare all functions before they are used.

Functions can be defined to return any type except a function or array. Functions can be defined to return pointers to other functions or arrays, however.

There is a fine distinction between a function declaration and a function definition. A function declaration tells the compiler that a function exists, either later in the same source file, in a separately compiled or assembled module, or in a library, but does not include the statements that tell the compiler what the function is supposed to do. A function definition includes a function body, which is a compound statement. To declare a function, follow the declaration with a semicolon, as shown above. To define a function, follow the declaration with the function body, like this:

```
void greet(void)

{
printf("Hello, world.\n");
}
```

Note that there is no semicolon after the closing parenthesis in the first line. It is an error to place one there.

Whenever a function is declared in a program, the compiler assumes that the definition occurs elsewhere. If the definition is not within the current program segment, you must use the extern storage class specifier, as in

```
extern void greet(void);
```

The storage class extern can also be used when the function definition will appear later in the same source file. The storage class static can also be used to declare a function that will be defined later in the same source file. The difference between the two is that, with the storage class static, the function must appear in the same source file, and static functions are not available outside the source file, while functions with a storage class of extern can be called from separately compiled modules.

The default storage class for a function is extern.

ORCA/C supports two special kinds of function definitions. The first is an inline definition, used to create interface files for the Apple IIGS tools. An inline declaration replaces the function body with the word inline, followed by two integer constants enclosed in parentheses and separated by a comma. The first integer is loaded into the X register, and then a jsl is performed to the second integer. The first integer, then, is the tool number of the tool to call, and the second integer is the main entry point into the toolbox, at 0xE10000. Allowing you to supply the second value means you can use the inline mechanism to call user tools as well as system tools. For examples of inline functions, see any of the tool interface files.

Another type of function is a function written entirely in assembly language. These functions have a return type and a parameter list, just as other functions do. The compiler uses the function return type and parameter list to check function calls in the rest of the program to make sure parameters are passed correctly, and to check to make sure that the value returned by the function is used in a legal way, but it is up to you to actually write the assembly language statements that use the parameters, remove them from the stack before returning to the caller, and to return any values to the caller.

An example of an assembly language function is shown below.

```
/* See if a key has been pressed; return 128 is so, and 0 otherwise */

asm int keypress ()

{
     lda    >0xC000
     and    #0x0080
     rtl
}
```

For a description of the syntax of assembly language statements, see the description of the asm statement in Chapter 18. For details on how parameters are passed and how function values are returned, see Chapter 5.

Parameters

In addition to returning a result, functions can take inputs in the form of passed parameters. C supports two different ways of handling parameters that have little to do with one another. The historical reason for this is that older C compilers use a very simple mechanism for defining parameters, while modern C compilers, including ANSI C compilers, support a parameter passing mechanism that allows the compiler to do some compile-time checking of function calls. These parameter mechanisms will be described separately.

Traditional C Parameters

```
parameter-list: identifier {',' identifier}*
```

The original parameter passing mechanism is very simple. Function declarations do not have parameter lists, even if the function that will be called allows or requires parameters. In a function definition, the names of the parameters are listed between the parentheses that follow the function name, separated by commas. The parameters are then defined before the start of the function body. Each parameter that appears in the parameter list must be defined, and, while types, structures, unions and enumerations may be defined, the only variables that can appear are those listed in the parameter list. As an example, the following function accepts an integer and a pointer to a string as parameters.

```
void roman(numeral, digit)

char *numeral;
int digit;

{
/* statements go here */
}
```

For each parameter there must be exactly one variable declaration between the function declaration statement and the body of the function. No other variables can be defined in this area, although type declarations are allowed. Parameters may not be initialized, and the only storage class that can be used is register. Parameters, like variables, can be any type except void or a function.

Because function declarations do not allow parameter declarations, traditional C parameters cannot be checked for correctness by the compiler. For example, if the function shown above is called using a statement like

```
roman(4);
```

there is obviously a problem: the function definition expects two parameters, while the call is only passing one.

Function Prototypes

```
parameter-type-list:  parameter-declaration {',' parameter-
     declaration}*
parameter-declaration:  declaration-specifiers
     [declarator abstract-declarator]
abstract-declarator:  {non-empty-abstract-declarator}
non-empty-abstract-declarator:     ['('  non-empty-abstract-declarator
     ')'] | [abstract-declarator '(' ')'] | [abstract-declarator '['
     {expression} ']'] | ['*' abstract-declarator]
```

Function prototypes correct the deficiencies in C described in the last section. They give the compiler the ability to check a function call to ensure that the parameters passed by a function call match the parameter list expected by the corresponding function. With a function prototype, each parameter is specified as a variable declaration, rather than simply a name. The function prototype can be used in both the declaration and definition of a function. Once a function has been declared or defined using a function prototype, the compiler checks subsequent calls to the function, flagging any calls that pass a parameter list that the function cannot handle as an error.

For an example, we will repeat the example from the last section using function prototypes.

```
void roman(char *numeral, int digit)

{
/* statements go here */
}
```

This simple example shows that the major difference in the way a function is defined using function prototypes is that the variable declarations are moved into the parameter list. Once the function has been declared or defined, however, the call

```
roman(4)
```

would cause the compiler to flag an error, rather than silently producing a program that might crash.

In addition to detecting parameter lists that have too few, too many, or incorrect types of parameters, function prototypes allow the compiler to do type conversions. With a traditional parameter list, for example, if you define a function that expects a long integer, and call it with an integer parameter, the result can be as severe as a run-time crash, and will rarely give a correct answer, even if the program does not crash. If the function has been declared using function prototypes, however, the integer parameter is converted to a long integer. All conversions that are performed during assignment using the = operator will also be performed for parameters passed to a function that has been declared using function prototypes.

A special case arises when a function has no parameters. The parameter list should include the single word void, indicating that no parameters are allowed. Any call to the function that tries to pass a parameter will then be flagged as an error.

```
/* prototype function with no parameters */
extern void MakeMyDay (void);
```

It is possible to use an abstract declarator in a function prototype. Basically, an abstract declarator defines a type without giving a variable name. This form of declaration is usually restricted to parameter lists for function declarations, rather than function definitions. An abstract declarator would be of no use in a function definition, since the parameter would have no name, and thus could not be referenced in the function body. In a function declaration, however, it allows you to tell the compiler about the parameter list without specifying the names of the parameters.

There is an important restriction that applies when using function prototypes. Functions must be declared before they are used. If a function is used before it is defined, the compiler sets up a default declaration. This declaration assumes that the function is not prototyped. Later, when the function is defined with a prototyped parameter list, a conflict arises, and the compiler flags an error.

Variable Length Parameter Lists

Many C functions, most notably those in the standard input and output library, use variable length parameter lists. For example, when you call printf, you always supply a format string, but you can also supply additional parameters. ANSI C introduced a mechanism to handle variable length parameter lists entirely from C, although the method requires you to use a function prototype.

Basically, the parameter list is split into a fixed part and a variable part. The fixed part is required: at least one fixed variable must be present. The variable part is represented in the function prototype by three periods, and must appear at the end of the parameter list. For example, to declare a function that will add one or more integers, returning the sum as a result, we would use

```
int sum(int first,...)
```

In this case, first is the fixed parameter.

The functions va_start, va_arg and va_end, from the stdarg.h library, provide a way of using the variable length parameter list from C. For examples of their use, see the description of va_arg in Chapter 19.

Variable argument lists will not work if stack repair code is enabled; stack repair code is enabled by default. For an explanation of stack repair code, see the next section. To turn off stack repair code, see the optimize pragma.

Common Mistakes With Parameters

Probably the most pervasive programming error in C programs is misuse of parameter lists. This isn't helped by the fact that many compilers seem to work with some kinds of parameter errors, and as a result, some programmers have written supposedly portable C programs that are incorrect. In all C standards, from the original Kerninghan and Ritchie specification right through to the most recent ANSI C standard, passing the wrong number or wrong type of parameters to a function gives, as the standards put it, undefined results. Basically, that means that a C compiler can support that practice if it chooses, but programs written that way are not portable to other C compilers; it is perfectly legal for a C compiler to simply ignore the possibility that the error can occur, resulting in a program that could actually crash. Passing a different number of parameters than were expected by the function being called was one early way to handle variable argument lists, which helped encourage the practice, but it is not supported by all C compilers.

To make this work, C compilers that support the practice of allowing you to call a function with the wrong number or type of parameters have the caller remove parameters from the stack after control returns from the function. In effect, this patches the stack, removing parameters whether or not the function that was called knew they were on the stack. On the 65816 CPU used in the Apple IIGS, it is a little more efficient for the function to remove its own parameters, and that is how ORCA/C works. As a result, if you call a function with the wrong number or type of parameters, you risk crashing the computer.

On the one hand, ORCA/C is a perfectly correct implementation of C, since passing incorrect parameters is not a feature that C compilers must support, but on the other hand, there are a lot of programs that assume this is legal C, and parameter errors are unfortunately easy to make in C. For that reason, by default, ORCA/C installs stack repair code that makes sure extra parameters are removed from the stack. This stack repair code takes up a lot of room and slows execution speed considerably, though, so ORCA/C also has a way to turn off the stack repair code. For more information on turning off the stack repair code, see the optimize pragma. Finally, to help you track down this sort of error, ORCA/C also has a debug option that will tell you when the parameter list is the wrong size; you can find a description of this feature under the debug pragma.

How Parameters are Passed

Integers, floating-point variables, and pointers are always passed by value. C does not have a mechanism for passing one of these types by reference like Pascal or Ada. This means that a function cannot change to original value of a variable passed as a parameter. For example, the program

```
void change(int i)

{
++i;
printf("%d\n", i);
}
```

```
int main(void)

{
int i;

i = 1;
printf("%d\n", i);
change(i);
printf("%d\n", i);
}
```

does not change the value of i in main. The program will print 1, 2 and 1 to the screen, not 1, 2 and 2.

This does not mean that there is no way to create a function that can modify a variable in another function, just that there is no way to do it directly. When a function is supposed to change the original value of a variable, a pointer to the variable is passed, rather than the variable itself. Using this idea in the example just presented, we can create a program that will print 1, 2 and 2:

```
void change(int *i)

{
*i += 1;
printf("%d\n", *i);
}

int main(void)

{
int i;

i = 1;
printf("%d\n", i);
change(&i);
printf("%d\n", i);
}
```

Arrays, structures and unions, on the other hand, are always passed by reference. In all three cases, the address of the first byte of the structure is passed, not the actual bytes that make up the structure. If the function that is called makes any change to the array, structure or union, the change affects the copy passed to the function. The following example illustrates this by using a function to clear an array.

```
void clear(float matrix[10][10]);

{
int i,j;

for (i = 0; i < 10; ++i)
   for (j = 0; j < 10; ++j)
      matrix[i][j] = 0.0;
}

int main(void)

{
float a[10][10];

clear(a);
/* a is now all zeros */
}
```

ORCA/C passes parameters starting with the rightmost parameter, and working to the left. The expressions are also evaluated in that order. While there is no strict requirement that parameters be processed this way in C, most C compilers follow the same practice. Simple variables of type char, short, int, long, float, double and extended are all passed by value, placing the actual value on the stack. The most significant byte is always placed on the stack first; since the 65816 stack builds from the top of memory towards the bottom, this means that the values appear least significant byte first in memory. Pointers are passed as unsigned long values, and will always have a range of 0 to 0x00FFFFFF. Arrays, structures and unions are passed by placing the address of the first byte of the array, structure or union on the stack. The compiler expects that all parameters are removed from the stack by the function called.

Returning Values from Functions

Functions can return any type except an array or function, although they can return a pointer to an array of function. The value, if any, is returned as an expression on a return statement. Functions returning void should use the return statement without an expression. If function is declared as returning void, any expression in the return statement will be flagged as an error.

Pascal Functions

C functions have two major differences from functions and procedures in other languages. First, most C compilers push the parameters to a function onto the stack working from right to left. In all other common languages, parameters are generally passed starting with the leftmost parameter and working to the right. Second, C is a case sensitive language, so that the functions foo and Foo are different. All other common languages are case insensitive, and would flag an error if you attempt to create two functions whose names differ only in the case of the letters used.

The pascal qualifier allows you to create C functions that can be called from other languages, or to call procedures and functions defined in languages other than C.

To use the pascal qualifier, place the word pascal immediately after the storage class specifier (if any), and before the function type. For example,

```
extern pascal int sum(int a, int b);
```

could be used to allow C to call a function named sum that requires two integer parameters and returns an integer result. Because the pascal qualifier has been used, the integer a is push on the stack before the integer b, exactly the opposite of the normal order. This function could be written in any language – including C, so long as the function definition also specifies the pascal qualifier.

The C compiler still treats the names of functions using the pascal qualifier as case sensitive, but the names are converted to uppercase characters before passing them on to the linker. This preserves the feel of C, but satisfies the requirements of other languages. It does create one problem, however: it is possible to define two functions in C whose names differ only in the case of the letters used, and end up with a linker error, while the compiler does not see a problem. If this happens, you must choose names that have differences other than the case of the letters.

Chapter 17 – Expressions

Syntax

In other chapters about the compiler, the syntax charts for the language is intermixed with the description of the C language. This doesn't work well for expressions, which can be explained more easily in terms of the operators and operands that make up the expression. This section presents the syntax charts for expressions for completeness.

```
expression:  comma-expression
comma-expression:  assignment-expression {',' assignment-expression}
assignment-expression:
     conditional-expression |
     [unary-expression
     [
     '=' | '+=' | '-=' | '*=' | '/=' | '%=' | '<<=' | '>>=' | '^=' |
     '='
     ]
     assignment-expression]
unary-expression:  postfix-expression
     | cast-expression
     | sizeof-expression
     | ['-' | '+' | '!' | '~' | '&' | '*' | '--' | '++'] unary-
     expression
postfix-expression:
     identifier |
     constant |
     ['(' expression ')']
     {['[' expression ']'] | ['.' identifier] | ['->' identifier]
      | ['(' {comma-expression} ')'] | '++' | '--'}*
cast-expression:  '(' type-name ')' unary-expression
type-name:  type-specifier abstract-declarator
sizeof-expression:  sizeof ['(' type-name ')'] | unary-expression
conditional-expression:
     logical-or-expression {'?' expression ':' conditional-
     expression}
logical-or-expression:
     logical-and-expression {'||' logical-and-expression}*
logical-and-expression:
     bitwise-or-expression {'&&' bitwise-or-expression}*
bitwise-or-expression:
     bitwise-xor-expression {'|' bitwise-xor-expression}*
bitwise-xor-expression:
     bitwise-and-expression {'^' bitwise-and-expression}*
```

```
bitwise-and-expression:  equality-expression {'&' equality-
     expression}*
equality-expression:
     relational-expression {['==' | '!='] relational-expression}*
relational-expression:
     shift-expression {['<' | '<=' | '>' | '>='] shift-expression}*
shift-expression:
     additive-expression {['<<' | '>>'] additive-expression}*
additive-expression:
     multiplicative-expression {['+' | '-'] multiplicative-
     expression}*
multiplicative-expression:
     unary-expression {['*' | '%' | '/'] unary-expression}*
```

Operator Precedence

C has a rich variety of operators, as well as a complicated set of operator precedences. Operator precedence is what forces the value of the expression 1+2*3 to be 7, rather than 9, which would be the result if the expression where evaluated left to right. The table below shows the precedence of all of the operators. The operators with the highest precedence are shown above the operators with lower precedence.

```
terms
++1    --1
++2    --2    sizeof (...)3 ~      !      -4     +4     &5     *6
*      /      %
+      -
<<     >>
<      >      <=     >=
==     !=
&7
^
|
&&
||
? :8
=      +=     -=     *=     /=     %=     <<=    >>=    &=     ^=     |=
,
```

1 postfix operators
2 prefix operators
3 type casts
4 unary addition, subtraction
5 address operator
6 indirection operator
7 bitwise and
8 conditional evaluation operator (a ternary operator)

Terms

Expressions are made up of terms separated by operators. The terms represent the variables and constants to be worked on, or the locations where these variables will be stored. This includes elements of arrays and components of structures and unions. Terms also include values returned by functions.

L-values

In discussing the various forms that a term can take, some will be referred to as l-values. This is an important concept in any language, but is much more important in C, with its multiple assignment operators and operators with side effects. Conceptually, an l-value is any simple value that can be changed. L-values include arithmetic variables, elements of arrays, structures, unions, components of structures and unions, enumeration variables and pointers. L-values do not include entire arrays, functions, or constants.

Constants

The simplest type of term is the constant. Constants can appear in base ten, base eight (octal) or base sixteen (hexadecimal). Constants can be coded as integers, real numbers, or strings, with a variety of attributes. Constants are not l-values. Constants are covered fully in Chapter 11.

Simple Variables

Another simple type of term is the variable. The simple variable can have a variety of types, but always appears as a name in the expression. The variable must be defined before it is used. Variables come in all of the same types that constants come it, except for strings: a string variable is actually an array of characters. Variables are l-values. All operators have some restrictions on the type of variables they can be used with.

Arrays

Arrays can appear in expressions in one of two ways, with or without a subscript. When an array appears without a subscript, it is not an l-value. It can appear in several places in this form. When an array appears in the operand of the sizeof operator, the size of the entire array is returned. For example,

```
int a[10];

...
```

```
    i = sizeof(a);
```

would set i to twenty, since the array a consists of ten integers, each of which is two bytes long. In other situations, the usual unary or binary conversions are performed. (The usual conversions are discussed in the section "Automatic Type Conversions," later in this chapter.) When the conversions are performed, the array is converted to a pointer to the first element of the array, and the type of the result is a pointer to an element of the array. For example,

```
    int a[10], *ip;

    ...

    ip = a;
```

sets ip to point to a[0].

An array element is accessed by coding the name of the array followed by a left bracket, an expression, and a right bracket. The expression is evaluated. The result type must be integral; it is used to index into the array. The result type of the entire term is the same as the type of one element of the array. The type of the term is an l-value if one term of the array is an l-value, and it is not an l-value if one element of the array is not an l-value. For example, with the definition

```
    int a[10][10];
```

a[1] is not an l-value, since it refers to a ten-element array of integers. A[1][3], on the other hand, is an l-value: the type of the term is int.

The subscript operator can also be applied to a pointer. Just as array names are treated as pointers to the first element of an array, a subscript operator applied to a pointer treats the pointer as the address of the first element of an array whose elements have the same type as the base type of the pointer. For example, the following statements show four ways to fill an array of ten integers with the numbers one through ten. The second method shows indexing of a pointer. Note the variety of other ways made possible by the close relationship between pointers and arrays in C.

```
    int a[10], i, *ip;

    /* method 1: subscripting the array */
    for (i = 0; i < 10; ++i)
       a[i] = i+1;

    /* method 2: subscripting the pointer */
    ip = a;
    for (i = 0; i < 10; ++i)
       ip[i] = i+1;

    /* method 3: dereferencing the pointer */
    for (i = 0; i < 10; ++i)
       *(ip+i) = i+1;
```

```
/* method 4: dereferencing the array */
for (i = 0; i < 10; ++i)
   *(a+i) = i+1;
```

Function Calls

A function call is coded as the name of a function followed by a pair of parentheses. If the function requires (or allows) parameters, the parameters are coded as expressions separated by commas, and appear between the parentheses. The parentheses are required even if the function has no parameters. The result type is the type returned by the function. It is not an l-value. If the function returns void, the result may not be used as an operand for any operator described in this chapter; this is equivalent to the procedure call or subroutine call in other languages.

If a function appears in an expression without the parentheses that indicate a parameter list, it is treated as a pointer to the function. This property of functions is often used when assigning a value to a pointer to a function, or when passing a function as a parameter. To call a function when a pointer to the function is available, code the pointer as if it were a function call. The following program uses these principles to print both the sine and cosine of π/4.

```
#define pi 3.1415926535

void print(float (*f)(), float val)

{
printf("%f\n", f(val));
}

int main(void)

{
print(sin, pi/4.0);
print(cos, pi/4);
}
```

Actual parameters to functions are evaluated starting with the rightmost parameter and proceeding to the left. The order in which parameters are evaluated varies from compiler to compiler. Since the order of evaluation can be critical to the outcome of functions with side effects, it is best not to use such functions in your programs. The following example illustrates this. It can also be used to test the order of evaluation used by a particular compiler.

```
int val;

int change(void)

{
return val++;
}
```

```
int test(int a, int b)

{
return a+b;
}

int main(void)

{
val = 1;

if (test(2*change(), change()) == 4)
   printf("Arguments are evaluated left-to-right.\n");
else
   printf("Arguments are evaluated right-to-left.\n");
}
```

Component Selection

The direct selection operator allows selection of a field from within a structure or union. It appears as a period separating an expression whose result is a structure or union (on the left) from the name of a component of the structure or union (on the right). The result has the same type as the field of the structure or union. It is an l-value if the expression to the left of the period is an l-value, and the component is not an array. The expression to the left of the period is not an l-value if the structure or union is returned by a function.

```
struct point {float x,y,z;};
struct polygon {point a,b,c,d;} p;

p.a.x = 1.0; /* set the x-coordinate of point a in polygon p to 1 */
```

The indirect selection operator, ->, is similar to the direct selection operator in that it is used to access components of structures and unions. The difference is that the left side is a pointer to a structure or union, rather than a structure or union. Once again, the result has the same type as the field of the structure or union. The result is an l-value if the component is not an array.

```
struct polygon *pPtr;

pPtr->a.x = 1.0; /* set the x-coordinate of point a 1 */
```

The indirect selection operator is completely equivalent to the direct selection operator applied to the pointer after it has been dereferenced. For example, the above example could be restated as

```
(*pPtr).a.x = 1.0;
```

and the result would not change.

Parenthesized Expressions

A parenthesized expression is an expression enclosed in parentheses. The parentheses do not affect the status of the enclosed expression in any way. In particular, the type of the parenthesized expression is the same as the type of the enclosed expression, and it is an l-value if and only if the enclosed expression is an l-value. The sole effect of the parentheses is to modify the precedence of the operators. The expression within the parentheses is evaluated before the surrounding operators are applied.

Postincrement and Postdecrement

The postincrement and postdecrement operators are used to increment or decrement scalar values. The operand must be an l-value. The result of the expression is the value before the operator is applied. For example,

```
i = 1;
j = i++;
printf("%d, %d\n", j, i);
```

prints 1, 2 to standard out, not 2, 2.

The result is not an l-value. The result type is the type of the operand.

If the ++ or -- operator is applied to a pointer, the updated pointer points to one object beyond (++) or before (--) the original value. The example below illustrates this by filling an array, then printing the contents backwards.

```
int i, a[10], *ip;

ip = a;
for (i = 1; i < 11; ++i)
   ip++ = i;

ip--;
for (i = 1; i < 11; ++i)
   printf("%d\n", ip--);
```

The result is technically undefined if an overflow or underflow results. In ORCA/C, if unsigned numbers are used, the result wraps around zero. For example, using unsigned integers, incrementing an integer whose value is 65535 would result in zero, and decrementing zero would yield 65535. For the case of signed integers, incrementing 32767 would give -32768, while decrementing -32768 would give 32767.

Math Operators

C supports a total of ten operators for dealing with numeric values. Five of these are binary operators, four are unary operators, and one is a built-in function. The operations, and the symbols used to represent the operations, are shown below.

operation	symbol	type	operands
addition	`+`	binary	any arithmetic type or pointer
subtraction	`-`	binary	any arithmetic type or pointer
multiplication	`*`	binary	any arithmetic type
division	`/`	binary	any arithmetic type
remainder	`%`	binary	any integer type
unary addition	`+`	unary	any arithmetic type
unary subtraction	`-`	unary	any arithmetic type
increment	`++`	unary	any arithmetic type or pointer
decrement	`--`	unary	any arithmetic type or pointer
size	`sizeof()`	function	any type or unary expression

Integer types include char, short, int, long, and the unsigned forms of all of these. Arithmetic types include the integer types plus float, double, comp and extended.

Addition

The operands are converted to the same type using the usual binary conversion rules. The two operands are then added. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

The addition operator can also be used to add an integer to a pointer. The result is a pointer of the same type as the input pointer. If, for example, n is added to a pointer, the new pointer points n elements past the original pointer. If ip is a pointer to an integer, then, *(ip+2) load the integer that is two integers past the integer pointed to by ip.

Subtraction

The operands are converted to the same type using the usual binary conversion rules. The second operand is then subtracted from the first. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

If the operands are one of the unsigned integer types, and the left operand is smaller than the right operand, the result is a positive unsigned number wrapped through the given base. For example, if the operands are unsigned int, and the left operand has a value of 1, while the right has a value of 2, then the result is 65536-1, or 65535. Another way of thinking about this is to recognize that this result has the bit pattern of the signed result of the operation, but that it is represented as an unsigned number.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

The subtraction operator can also be used with two pointer operators if the pointers are of the same type. The result is an integer; it is the number of elements between the two pointers. For example, (&a[4])-(&a[1]) would be 3, regardless of the type of the array.

Multiplication

The operands are converted to the same type using the usual binary conversion rules. The two operands are then multiplied. The result is the same type as the converted operands, and is not an l-value.

Integer overflow is not detected. If two integers exceed the range of the type of the operands, the extra most-significant bits are lost.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

Division

The operands are converted to the same type using the usual binary conversion rules. The first operand (the numerator) is then divided by the second operand (the denominator). The result is the same type as the converted operands, and is not an l-value.

For integer division, any fractional part of the result is discarded; i.e., 4/3 gives a result of 1, not 1.333. For positive results, all C compilers return the truncated number; i.e., the largest integer that is less than or equal to the floating-point result. Various compilers truncate negative numbers in different ways, so it is not a good idea to depend on the results of integer division when the results are negative. In ORCA/C, as in most implementations of C, truncation of a negative result returns the integer closest to zero. For example, (-4)/3 gives a result of -1, not -2.

Integer division by zero is not detected. The results of dividing by zero are not predictable.

For floating-point division, the result is the floating-point number that is the best representation of the correct answer; here, some differences between the actual result and the correct mathematical result may occur due to round-off error.

Floating-point errors are not detected automatically. Errors can be detected by direct calls to SANE. If a floating-point underflow occurs, the result is zero. Floating-point overflow gives a result of infinity, which can lead to correct, non-infinite answers in some forms of equations.

Remainder

The operands are converted to the same type using the usual binary conversion rules. The first operand is then divided by the second. The result is the remainder from the division. The result is the same type as the converted operands, and is not an l-value.

For non-zero b, the following relation always holds. It defines the action of this operation in a mathematical sense, including the results when one argument is negative and the other is positive.

```
((a/b)*b + a%b) == a
```

Integer division by zero is not detected. The results of dividing by zero are not predictable.

The remainder function requires integer operands; it is an error to supply a floating-point value as one or both of the operands.

Unary Subtraction

The operand, which can be of any arithmetic type, is converted using the usual unary conversion rules. The result is not an l-value. The operation is completely equivalent to subtracting the value from zero. The same rules that would apply to subtracting the number from zero also apply to unary subtraction when handling overflows, underflows, and dealing with unsigned operands.

Unary Addition

The operand, which can be of any arithmetic type, is converted using the usual unary conversion rules. The result is not an l-value. The operation is completely equivalent to adding the value to zero. The same rules apply for handling overflows, underflows, and dealing with unsigned operands. Other than any type conversions performed, this operation does not actually generate any code, since adding a number to zero does not change the number.

Prefix Increment

The operand, which can be and scalar l-value, is incremented by one. The usual binary conversions are applied to the operand and to the constant one. The result is stored back in the operand after the usual assignment conversions are applied. The result is the new operand, and is not an l-value. The type of the result is the same as the type of the operand.

If the operand is a pointer, the pointer is moves so that it points to the next item of the type pointed to by the pointer. For example, if the pointer is a pointer to an integer, a value of two is added to the ordinal value of the pointer, since integers are two bytes long.

If the argument is an unsigned value, and the value of the argument is the largest unsigned number that can be represented with the given integer size, the result is zero. If the argument is a signed integer and the value is the largest signed integer, the result is technically undefined, but is actually one less than the value of the argument subtracted from zero in ORCA/C. For example, if the value if the signed integer i is 32767 (the largest signed integer), and the ++ operator is applied to the integer, the result is -32768.

Prefix Decrement

The operand, which can be and scalar l-value, is decremented by one. The usual binary conversions are applied to the operand and to the constant one. The result is stored back in the operand after the usual assignment conversions are applied. The result is the new operand, and is not an l-value. The type of the result is the same as the type of the operand.

If the operand is a pointer, the pointer is moves so that it points to the previous item of the type pointed to by the pointer. For example, if the pointer is a pointer to an integer, a value of two is subtracted from the ordinal value of the pointer, since integers are two bytes long.

If the argument is an unsigned value, and the value of the argument is zero, the result is the largest unsigned number that can be represented with the given integer size. If the argument is a signed integer and the value is the smallest signed integer, the result is technically undefined, but is actually one less than the value of the argument subtracted from zero in ORCA/C. For example, if the value if the signed integer i is -32768 (the smallest signed integer that can be represented in two bytes using two's complement notation), and the -- operator is applied to the integer, the result is 32767.

Sizeof Operator

The sizeof operator returns the size of the operand, in bytes. The result is not an l-value. The operand can be a type name or any unary expression. The operand cannot be a function, void, or an array declared without giving explicit values for all dimensions. If the operand is an array, the result is the complete size of the array. For any other unary operand, the result is the size of the result type.

There are two forms of the sizeof operator. When the operand is a type name, it must be enclosed in parentheses. When the operand is a unary expression, it does not have to be enclosed in parentheses, although it does no harm to use the parentheses.

When the operand is a unary expression, the expression is not evaluated at run-time. In other words, ++ and -- operators do not change the value of the variables, function calls are not made, and assignments are not performed. The expression is examined at compile time to determine the result type, but no code is generated.

Taking the size of a bit field returns the size of the underlying type.

While the sizeof operator does not perform any type conversions, the expression appearing in the operand can perform type conversions.

In ORCA/C, the result of the sizeof operator is of type unsigned long.

The examples below illustrate some of these principles. The declarations which precede the tables are used to compute the sizes of the terms.

```
int *ip, i, a[10];
float f, fa[5][5];
struct point (float x,y,z;} p;

sizeof (int)     2
sizeof (char)    1
sizeof (unsigned)       2
sizeof (long)    4
sizeof (float)   4
sizeof (double)  8
sizeof (extended)       10
sizeof (comp)    8
sizeof (void *)  4
sizeof (i)       2
sizeof ip 4
sizeof (*ip)     2
sizeof a  20
sizeof a[1]      2
sizeof f  4
sizeof fa 100
sizeof (point)   12
sizeof p  12
sizeof (0L)      4
sizeof (1+7)     2
sizeof (sizeof(0))      4
```

Comparison Operations

There are six comparison operators in C. The operands of any of these operators can work on any arithmetic type, or on two pointers if both pointers are of the same type. The equality operators can also be used on a pointer and the integer constant zero. Two pointers are considered to be equal if they point to the same byte of memory, or if both pointers are NULL. A pointer is equal to the integer constant zero if the value of the pointer is NULL. Pointer a is less than pointer b if a points to an object stored at a smaller address than the object b points to. The only time this is generally of concern in a C program is when the pointers point to elements of the same array. In that case, pointer a is less than pointer b if pointer a points to an element with a smaller subscript than the element pointed to by b.

The six comparison operators are shown in the table below.

operator	condition
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	equal
!=	not equal

For example, a < b is true if a is less than b, and false if it is not.

If the condition tested by the operator is true, the result is a signed integer value of one. If the condition is false, the result is zero. The result is not an l-value.

Logical Operations

There are two binary logical operators, and one unary logical operator. All of the operations accept any scalar type as an operand. The result is of type integer, and is one for a true result, and zero for false. The result is not an l-value.

The logical and operator is &&. The first operand is evaluated. If it is non-zero, it is treated as true, while a zero result is treated as false. If the result is false, the right operand is not evaluated at all, since the result has already been determined, and the result of the && operation is zero. If the left operand is non-zero, the second operand is evaluated. If it is zero, the result is zero (false); if the second operand results in a non-zero value, the result is one (true).

The logical or operator is ||. The first operand is evaluated. If it is non-zero, it is treated as true, while a zero result is treated as false. If the result is true, the right operand is not evaluated at all, since the result has already been determined, and the result of the || operation is one. If the left operand is zero, the second operand is evaluated. If it is zero, the result is zero (false); if the second operand results in a non-zero value, the result is one (true).

The logical negation operator is !. The single operand, which can be of any scalar type, is converted using the standard unary conversion rules. If the operand is zero, the result is one; if the operand is non-zero, the result is zero.

Bit Manipulation

C has six powerful bit manipulation operators. The operands for all of the operators can be of any integer type. The result is not an l-value.

The bitwise and operator is &. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an and of the individual bits in the two operands. For example, 0x1248 & 0x1441 == 0x1040.

The bitwise or operator is |. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an or of the individual bits in the two operands. For example, 0x1040 | 0x0208 == 0x1248.

The bitwise exclusive or operator is ^. It is a binary operator. The two operands are converted using the standard binary conversion rules. The type of the result matches the type of the converted operands. The result is formed by performing an exclusive or of the individual bits in the two operands. The result of an exclusive or is 1 if one of the bits, but not both, is 1, and 0 of both bits are 0 or both bits are 1. For example, 0x1248 ^ 0x1040 == 0x0208.

The bitwise negation operator is ~. It is a unary operator. The single operand, which appears to the right of the operator, is converted using the standard unary conversion rules. The type of the result matches the type of the converted operand. The result is formed by performing an exclusive or of the individual bits in the operand with a second operand composed entirely of ones. The effect is to reverse each bit in the operand. For example, ~0x1248 == 0xEDB7.

C supports two binary shift operators, << and >>. In each case, the operands are converted separately using the standard unary conversion rules, not the binary conversion rules. The result type is the type of the converted left operand. It is formed by shifting the operand a certain number of bits to the left (for the << operator) or right (the >> operator).

For unsigned operands, zeros are shifted in from both the left and right to fill the new bit positions. Zeros are also used to fill the bit positions created by the shift left operator (<<) if the left operand is signed. If the shift right operator (>>) is used with a signed left operand, the sign bit is replicated to fill the unused bit position. The result of these rules is that shifting a number left is mathematically equivalent to multiplying the number by two raised to the power of the second operand, providing that no overflows occur. Shifting an unsigned number to the right is equivalent to dividing the number by two raised to the power of the second operand. The same is true for signed operands unless the result would be zero; in that one case, the result of shifting is always -1.

If the right operand is zero, the result is the value of the left operand. The results are not predictable if the right operand is less than zero. Even if you determine the results by experimentation, we reserve the right to change the compiler in the future in ways that could affect such results.

Assignment Operators

Simple Assignment

C has a variety of assignment operators. The simplest is the assignment operator, =. The simple assignment operator evaluates the expression to the right of the operator, then assigns the result to the l-value that appears to the left of the operator. The result of the expression is the value assigned to the l-value, and has the same type as the unconverted l-value.

The simple assignment operator can be used with any arithmetic types. If the type of the expression does not match the type of the l-value, the expression is converted to the l-value's type using the same rules that would apply if the expression were to be cast to the correct type explicitly. See the discussion of type casing for details.

Structures and unions can be assigned so long as the type of the expression matches the type of the l-value exactly. The contents of the structure or union are copied to the destination structure or union. All bytes are copied, even if some involve unused bit fields, bits used to align other fields to a byte boundary, or unused bytes in a union. The number of bytes copied is equal to the number of bytes reported by the sizeof operator for the size of the structure or union.

Pointers can be assigned so long as the type of the expression matches the type of the l-value exactly. Some C compilers automatically cast pointer types during assignment; ANSI C does not permit automatic casting of pointer types. It is also legal to assign an integer to a pointer so long as the integer is a constant with a value of zero.

Finally, an array can be assigned to a pointer if the elements of the array are of the same type as the values pointed to by the pointer. The resulting pointer points to the first element of the array.

The assignment operator cannot be used to copy one array into another.

Compound Assignment

The compound assignment operators are a combination of simple assignment and one of the arithmetic or bitwise binary operators. The effect is to perform the operation on the l-value and the expression, assigning the result to the l-value. The result is not an l-value. It has the same type as the l-value that appears to the left of the operation.

The one difference between a compound assignment operator and an equivalent expression formed using the simple assignment operator and the binary operation is that the l-value is only evaluated one time. For example, if a function is used to compute the subscript of an array, as in

```
a[f(x)] += 3;
```

the function f is only called one time. The equivalent expression using the simple assignment operator is

```
a[f(x)] = a[f(x)] + 3;
```

In this case, the function f is called two times. While it is rare in well-written programs, there are cases where the two expressions would yield different results.

The same restrictions that apply to the operator associated with the compound assignment also apply to the compound assignment operator. Errors are handled the same way, and the same types of operands are allowed. The table below shows the compound assignment operators with the associated binary operator and the types of operands that are allowed.

assignment operator	binary operator	operands
+=	+	any arithmetic type or pointer += integer
-=	-	any arithmetic type or pointer -= integer
*=	*	any arithmetic type
/=	/	any arithmetic type
%=	%	any integer type
<<=	<<	any integer type
>>=	>>	any integer type
&=	&	any integer type
^=	^	any integer type
|=	|	any integer type

The original C language allowed the operator to appear after the = character. This is no longer allowed in C, and most programs have been converted to avoid this form of the compound assignment operator. A more recent change is to require the compound assignment operators to be treated as single tokens. The effect of this change is that white space may not appear between the operator and the = character. This is a more recent change to the language, and some programs ported form other compilers may still have white space after the operator. The compiler will identify this error when it occurs, and the program can be easily corrected.

Multiple Assignments

Unlike the other binary operators in C, all of the assignment operators are right-associative. This means that if more than one assignment operator appears in a single expression, the rightmost operation is performed first. This allows for multiple assignments in one expression. For example, x, y and z can all be initialized to zero at one time, as shown below.

```
x = y = z = 0.0;
```

In most other high-level languages, this same operation would require three assignment statements.

Pointers and Addresses

The address operator & is used to obtain a pointer to an l-value. The result is a pointer of type "pointer to value," where value is the type of the l-value. For example, the following code uses the address operator to initialize a pointer to point to an integer.

```
int i, *ip;
```

```
ip = &i;
```

The address operator can be used with register variables in ORCA/C. Some C compilers do not permit the address operator to be used with register variables, and others may generate less efficient code if the address operator is used with a register variable.

When the address operator is applied to a function, the result is of type pointer to function. When the address operator is used on an array whose elements are of type T, the result is of type pointer to array of T, or simply pointer to T.

The address operator cannot be used on a bit field. For example, the following use of the address operator is illegal.

```
struct {unsigned i: 8; unsigned j: 8;} bar;
char *cp;

cp = &bar.i;     /* illegal */
```

The indirection operator is used to dereference pointers. It appears as an asterisk to the left of an l-value of type pointer. If the pointer is of type "pointer to T," the result is an l-value of type T.

The pointer must point to a valid variable when it is dereferenced. If it does not, the results are unpredictable. In most cases, a random value will be loaded, but if the pointer is pointing into one of the memory mapped I/O areas, almost anything can happen, including turning disk drive motors on or off, switching displays, and so forth.

Sequential Evaluation

The sequential evaluation operator is a comma. It can be used between two expressions in many places where an expression can be used. The left expression is evaluated first. The result is discarded, and the right expression is evaluated. The result and result type are the same as the result and result type of the right expression. The result is not an l-value.

More than one comma operator can be used in a single expression, in which case the expressions are evaluated left to right, and the result and result type are determined by the last expression.

The comma operator cannot be used in parameter lists to function calls, field length expressions in structure and union delcarator lists, enumeration value expressions in enumeration lists, or initializer expressions in declarations and initializers. Parentheses can be used to allow the use of the comma operator in these situations.

Conditional Expressions

C has one ternary operator which can be used to conditionally execute a statement.

```
test() ? doTrue() : doFalse();
```

The first operand can be any scalar type. It is evaluated. If the first operand is non-zero, the second expression is evaluated. If the first operand is zero, the last expression is evaluated. The result is the evaluated second or third operand, and is not an l-value.

The last two expressions can take on several forms. If they are both arithmetic expressions, the usual binary conversion rules are applied, and the result is the common type, regardless of which expression is actually evaluated. They can also be pointers to the same type, in which case the result is a pointer to the same type. They can be structures or unions if the types are the same, in which case the result is the same type as the operand. Both expressions can be of type void, in which case the result is of type void. Finally, one can be the integer constant zero, and the other can be a pointer type. In that case, the result type matches the pointer.

The conditional operator is right-associative, so the expression

```
a ? b : c ? d : e
```

is interpreted as

```
a ? b : (c ? d : e)
```

Automatic Type Conversions

Assignment Conversions

When one value is assigned to another, the expression that appears to the right of the assignment operator must be of the same type as the l-value that appears to the left of the operator.

There are several cases when the compiler can automatically perform type conversions on the right hand side to force the type to match the type of the l-value. These are:

1. The integer 0 is assigned to a pointer. In that case, the integer is converted to a long, unsigned value and cast to the appropriate pointer type before storage.

2. The l-value and expression are of different arithmetic types. (The arithmetic types include all integer and floating-point types, as well as enumerations.) In that case, the expression is converted to the type of the l-value as if the expression were explicitly cast to the correct type. See the section on type casing for details.

There are many other conversions that are performed as a normal part of evaluating an expression. These conversions are discussed in the sections that follow.

Function Argument Conversions

When parameters are passed to a function for which a prototype exists, the expressions are converted using the same rules that would apply if the expression were being assigned to a variable of the type of the parameter. If no prototype exists, or if the parameter appears as an optional parameter in a variable length parameter list, the unary conversion rules are applied to the expression.

Unary Conversion Rules

The unary conversion rules are applied to the operands of the unary operators !, -, +, ~ and *. They are also applied to the leftmost operand of the ternary ? : operator, the operands of the bit shift operations << and >>, and to parameters to functions when function prototypes are not used.

Unary conversions are used primarily to reduce the number of forms an operator must handle. Short integers and char variables are converted to type int; unsigned short and unsigned char are converted to type unsigned. Float and double values are converted to type extended. Finally, arrays and functions are converted to pointers of the appropriate type.

Binary Conversion Rules

The binary conversion rules are used to ensure that both operands of a binary operator are of the same type before the operation is performed. The binary conversion rules are best expressed as a series of rules which are applied in turn until one of the conditions specified by a rule is matched. Before these rules are applied, the unary conversion rules are applied to each operand. The binary conversion rules, in the order in which they are applied, are:

1. If either operand is not an arithmetic type, or if the operands are of the same type, no conversion is performed.
2. If one operand is of type extended, then the other operand is converted to extended.
3. If one operand is of type double, then both operands are converted to extended.
4. If one operand is of type float, then both operands are converted to extended.
5. If one operand is of type unsigned long, the other operand is converted to unsigned long.
6. If one operand is of type long, the other operand is converted to long.
7. If one operand is of type unsigned int, the other operand is converted to unsigned int.

ORCA/C will convert all expressions involving a floating-point operand to type extended. While C traditionally requires this type of conversion, and SANE and the 68881 floating-point processor both make it economical, ANSI C does not require all floating-point calculations to be performed in the longest format available.

In the cases where the internal representation of the number remains the same, no actual code is generated by the compiler. In cases where the internal representation does change, the conversions are discussed in detail in the next section, type casting.

Type Casting

A type cast takes the form of a type name in parentheses immediately before an expression. The expression is evaluated, and then the value of the expression is changed to match the type specified.

Converting Integers to Integers

When one integer is converted to another, the new value is the same as the old if it can be represented in the new type.

There are several cases where the value of the original integer cannot be represented in the new type. The first occurs when a signed integer is converted to an unsigned integer of the same size, and the original value is less than zero. In that case, the new value is 2n+val, where n is 8 for unsigned char; 16 for unsigned short and unsigned int; and 32 for unsigned long; and val is the original value. Another way of looking at the conversion is that the bit pattern does not change, and the signed value becomes the unsigned equivalent of the two's complement bit pattern used to represent the number.

When an unsigned integer is converted to a signed integer of the same size, no conversion is performed. If the unsigned value is too large to represent in the signed form, the result is undefined. In fact, the result becomes the negative number whose bit pattern is the same as the original unsigned number.

If a shorter value is converted to a longer one, the only case where the arithmetic value cannot be represented is if the shorter value is a signed number, and the value is less than zero, and the longer number is an unsigned number. In this case, the number is first converted to a signed value of the same size as the unsigned type, and then the above rules apply.

If a long value is converted to a shorter one, and the final value cannot be represented exactly, the extra bits are discarded.

Converting Floating-Point Values to Integers

Floating-point values are converted to integers by discarding any fraction part, then converting the resulting number to an integer. If the number is too large or too small to represent as an integer, the results are unpredictable. Any of the floating-point formats can be converted to any integer format.

Converting Pointers to Integers

In ORCA/C (but not in all implementations of C), pointers are the same size as unsigned long integers. When converting a pointer to an integer, the compiler treats the pointer as if it were an unsigned long integer. If the pointer is converted to a format other than unsigned long integer, the rules for converting an unsigned long integer to the specified format apply.

Pointers can be converted to long and unsigned long with no loss of information. In general, converting a pointer to char, int or short and then converting the result back to a pointer will result in a different, probably incorrect, pointer value.

Converting Floating-Point Values to Other Floating-Point Formats

When converting from float to double or extended, or when converting from double to extended, there is no loss of precision, regardless of the input value.

When converting from extended to double or float, or when converting from double to float, several things can happen. It is possible that the value being converted cannot be represented accurately. For example, 1.00000000001 can be represented reasonably well using double or extended, but if the double or extended number is converted to float, loss of precision will result in a float value of 1.0. It is also possible that the exponent will be too large or too small for the smaller format. If the exponent is too small, underflow results, and the new number is set to zero. For example, converting the double number 1e-300 to a float number would result in zero. If the exponent is too large, overflow occurs, and the result is infinity. For example, converting 1e300 from double to float would cause an overflow.

Converting Integers to Floating-Point Values

All of the integer formats can be converted to any of the floating-point formats. In some cases, there may be a loss of precision. For example, float variables are accurate to about seven decimal digits, so converting the long integer 1000000001 to a float value would result in 1.0e9, not 1.000000001e9.

Converting to and from Enumerations

For the purpose of conversions, enumeration constants are treated as signed decimal integer constants, and enumeration variables are treated as type int.

Converting Pointers to Pointers

Any pointer type can be converted to any other pointer type. There is no loss of precision or change in representation, so converting the pointer back to its original type will always result in the original value. This is not true in all C compilers, so avoid making use of this principle in programs that will be ported to other machines.

Converting Integers to Pointers

The integer 0 is used to represent the null pointer. It can be converted to a pointer in all implementations of C.

ORCA/C also allows conversion of any other integer value to a pointer. To make sense on the 65816 CPU, the apparent range of the integer should be in the range 0x00000000 to 0x00FFFFFF. If it is not, using the pointer will generally result in the most significant byte being ignored. Some tools, however, may crash or access other areas of memory.

No Apple IIGS actually has all of the memory represented by these values installed. It is up to you to ensure that the integer value represents an area of memory that is safe to access. If the memory does not belong to your program, changing it will cause unpredictable results. These results could include damaging disk files or crashing the computer.

Converting Arrays to Pointers

An array name is automatically converted to a pointer to the first element of the array in all cases except when the array appears as the operand for the sizeof operator.

Converting Functions to Pointers

Except in cases where a function name is used to call a function, the function name is automatically converted to a pointer to the function.

Converting to Void

Any type can be converted to void, although the result cannot be used. The only place where this conversion is generally used is when a function that normally returns a value is called in a context where the return value is not needed. In that case, the conversion serves as a clue to the compiler that the result will not be used. This has no effect in ORCA/C programs, since the code to use a value returned by a function is not generated unless the value is actually used.

Chapter 18 – Statements

Compound Statements

```
compound-statement: '{' {initialized-declaration ';'}* {statement}*
   '}'
statement:
   [expression ';'] |
   labeled-statement |
   compound-statement |
   if-statement |
   while-statement |
   do-statement |
   for-statement |
   switch-statement |
   break-statement |
   continue-statement |
   return-statement |
   goto-statement |
   segment-statement |
   asm-statement |
   null-statement
labeled-statement:
   [identifier | [case expression] | default] ':' statement
```

Compound statements (also called blocks) are used as function bodies and to group statements together. For example, the for loop loops over a single statement. By using a compound statement, more than one statement can be executed each time the body of the loop is executed.

The compound statement has two parts. The first part is a series of declarations. If present, variables declared in the block constitute a new scope. They cannot be used or modified from outside of the compound statement; in fact, they do not exist in memory until the compound statement is entered. These variables can be initialized. If so, the initializations are only carried out if the compound statement is executed from the beginning. If control is passed to a statement in a compound statement by a goto statement or switch statement, the variables are not initialized. Since the variables in a compound statement have a scope limited to the compound statement, names that have been used globally, as parameters, or in the compound statement that the new one is embedded in can be redefined within the compound statement. If this is done, the declaration within the compound statement has precedence until control leaves the compound statement. For example, the following program is legal in C, and will print 1, 2, 1 to the screen, not 1, 2, 2.

```
int main(void)

{
int i;

i = 1;
printf("%d\n", i);

   {
   int i;

   i = 2;
   printf("%d\n", i);
   }

printf("%d\n", i);
}
```

The second part of the compound statement is a list of zero or more statements, including compound statements. Unless a statement changes the flow of control, these statements are executed one at a time until all of the statements have been executed. If the compound statement is the body of a function, control will return to the caller. If the compound statement is embedded in another compound statement, control will pass to the statement immediately after the compound statement.

It is legal to leave a compound statement using a goto, break, continue or return statement. It is also legal to use a goto or switch statement to enter a compound statement without executing some of the statements, although variables will not be initialized in that case.

Null Statements

```
null-statement:  ';'
```

Anywhere a statement can be used, a semicolon can be used. This is the null statement, which is a statement that takes no action and generates no code. It is generally used in connection with loop statements that have no statement body, such as a loop waiting for a interrupt to take place. A null statement is shown as the body of a while statement in the example below.

```
while (NoBurglar()) ;
```

Expression Statements

An expression followed by a semicolon can be used as a statement. The expression is evaluated, and the result, if any, is discarded.

While Statement

```
while-statement:  while '(' expression ')' statement
```

The while statement consists of the reserved word while followed by an expression in parentheses and a statement. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) != 0 is legal. If the expression is non-zero, the statement is executed, and the process repeats. If the expression evaluates to zero, control passes to the statement following the while statement.

The while loop can terminate early due to the effects of a return, goto or break statement.

```
/* initialize an array */
i = 0;
while (i < 10)
   a[i++] = 0;
```

Do Statement

```
do-statement:  do statement while '(' expression ')' ';'
```

The do statement consists of the reserved word do, a statement, the reserved word while, and an expression enclosed in parentheses. The statement is executed. Next, the expression is evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) != 0 is legal. If the result is zero, control passes to the statement following the while clause that concludes the do statement. If the result of the expression is non-zero, the process repeats.

The difference between the do statement and the while statement is that the do statement always executes the statement at least one time. The while statement checks the loop condition first; if it is zero, the statement never gets executed.

The do loop can terminate early due to the effects of a return, goto or break statement.

```
do
   printf("Please press a key.\n");
while (! KeyPress());
```

For Statement

```
for-statement:  for '(' {expression} ';' {expression} ';'
     {expression} ')' statement
```

The for statement is designed for use when a statement must be executed for a specified number of times. Because of its design, it can be used in many other situations. It consists of the reserved word for, followed by three expressions enclosed in parentheses and separated by semicolons, and a statement.

Each of the expressions is optional. Execution of the for statement starts by evaluating the first expression, which is normally used to initialize a loop variable. The second expression is then evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) != 0 is legal. If the result is non-zero, the statement is executed. The third expression, which is generally used to increment the loop counter, is then evaluated, and the process repeats starting at the evaluation of the second expression. If the second expression evaluates to zero, execution continues with the statement following the body of the for statement.

If the second statement is not coded, the compiler assumes that is always evaluates to a non-zero value.

```
/* initialize an array */
for (i = 0; i < 100; ++i)
   a[i] = i;
```

Another way to think of the for statement is that it is roughly equivalent to a while statement with the expressions placed in certain locations. The for statement

```
for (exp1; exp2; exp3) statement;
```

is equivalent to the statement

```
{
exp1;
while (exp2) {
   statement;
   exp3;
   }
}
```

The one exception is the way the continue statement works. In the for statement, the continue statement jumps to exp3; in the while statement, the continue statement jumps to the end of the block, just past exp3.

The for loop can terminate early due to the effects of a goto, return or break statement in the body of the loop.

If Statement

```
if-statement:  if '(' expression ')' statement {else statement}
```

The if statement is used to conditionally execute a statement. The expression is evaluated. The expression must be an arithmetic type; in particular, it must be of a type such that the comparison (expression) != 0 is legal. If the result is non-zero, the statement following the condition is executed. If the result is zero, the statement is not executed.

The optional else clause is used to provide a second statement that will be executed if the conditional expression evaluates to zero. It is not executed if the expression yields a non-zero result.

```
for (i = 1; i <= 10; ++i)
   if (i & 1)
      printf("%d is odd.\n", i);
   else
      printf("%d is even.\n", i);
```

Goto Statement

```
goto-statement:  goto identifier ';'
```

The goto statement is used to transfer control to another statement. The identifier is a named label. The label must appear on exactly one statement somewhere in the current function body. The next statement executed is the statement that the label appears on.

It is legal to place a label on a statement and not place a goto statement in the function that refers to that label. It is not legal, however, to try to branch to a label that does not exist, or to place two labels with the same name in the same function, whether or not a goto statement exists that refers to the duplicate label.

Switch Statement

```
switch-statement:  switch '(' expression ')' statement
```

The switch statement is used to choose from a list of statements. The expression, which must yield an integer value, is evaluated. The statement that follows is generally a compound statement. Conceptually, it is scanned for a case label with a value that matches the value generated by the expression. If such a value is found, control is passed to the statement following the case label. If there is no match, and there is a default label, the statement after the default label is executed. If there is no match, and there is no default label, control passes to the statement after the switch statement.

There are some restrictions on the labels in the switch statement body. The case labels must have constants that are the same type as the expression after the usual unary conversions are applied to both. For example, it is not legal to use a case label with an int constant if the expression results in a long int value. Duplicate labels are not permitted. This means that two case statements cannot have constant expressions that result in the same value, and two default labels cannot appear in the body of the statement. Case labels and the default label can only appear in the body of a switch statement.

After control is passed to one of the statements in the switch statement, execution continues as if a goto had been used to branch to the statement. For example, the following statements would print five lines on the screen. The first would have five asterisks, the second would have four, and so on. The point is that in some languages, execution would be transferred out of the switch statement when the next case label was encountered; this is not true in C.

```
for (i = 1; i <= 5; ++i) {
   switch (i) {
      case 1: printf("*");
      case 2: printf("*");
      case 3: printf("*");
      case 4: printf("*");
      case 5: printf("*");
      }
   printf("\n");
   }
```

As the example shows, execution falls through from the statements marked by one case label to the next group of statements. It is customary in C to use the break statement at the end of each of the groups of statements.

```
for (i = 1; i <= 5; ++i) {
   switch (i) {
      case 1: printf("*");
              break;
      case 2: printf("**");
              break;
      case 3: printf("***");
              break;
      case 4: printf("****");
              break;
      case 5: printf("*****");
      }
   printf("\n");
   }
```

The goto and return statements can also be used to leave the body of the switch statement before the last statement is executed.

Break and Continue

```
break-statement:  break ';'
continue-statement:  continue ';'
```

The break and continue statements are used to exit a loop early or branch to the end of the loop body.

The break statement can be used in the body of a while, do, for or switch statement. When it is encountered, control is transferred to the statement after the while, do, for or switch statement. For example, break statements can be used to cause the C switch statement to work like the Pascal case statement, as shown in the section above.

The continue statement is used to branch to the end of a while, do or for loop. Conceptually, the continue statement does a goto to the end of the loop body.

The easiest way to understand the break and continue statements completely is to look at the equivalent goto statements. The following four statement models show the statements which use break and continue statements. In all of the statements except switch, there is a label called C. A continue statement in the body of the loop (called body in the examples) is equivalent to a goto to the label C. In all of the statements, there is also a label called B. A break statement in the loop body is completely equivalent to goto B.

```
while (expression) {body; C: ;} B: ;
do {body; C: ;} while (expression); B: ;
for (expression; expression; expression) {body; C: ;} B: ;
switch (expression) {body;} B: ;
```

If a break or continue statement appears within nested statements, it applies to the most recent enclosing statement. In the following code fragment, continue and break statements appear inside a switch statement, which is in turn inside a for loop. The break statement exits the switch statement. Since the continue statement has no meaning in the switch statement, it applies to the for statement.

```
for (i = 0; i < 10; ++i)
   switch (i) {
      1: 2: 3: 5: 7:
         printf("%d is prime.\n", i);
         continue;
      9:
         printf("%d is odd.\n", i);
         break;
      default:
         printf("%d is even.\n", i);
      }
```

Return Statement

```
return-statement:  return {expression} ';'
```

The return statement returns control to the function that called the current function. If the current function is main, control is returned to the program launcher that was used to execute the program.

If the return statement is followed by an expression, that expression is evaluated. The type of the expression must be compatible with the type of the function in the same sense that an expression must be compatible with the l-value when the assignment operator is used. The value of the expression is returned to the caller as the return value of the function.

If a function has a return type, but control is returned by a return statement that does not have an expression, the returned value is unpredictable.

If the function executes to the end of the compound statement without encountering a return statement, the effect is the same as if a return statement with no expression were encountered.

Segment Statement

```
segment-statement:  segment string-constant {',' dynamic} ';'
```

The segment statement is used to break a program into two or more load segments. It can be used with either the large or small memory model. The affect is to cause all functions defined after the segment statement to appear in a new load segment. To understand what this does, we will look at the memory models used on the Apple IIGS in detail.

In any program compiled by ORCA/C, memory is allocated for up to five different purposes. Local variables are allocated from a stack frame allocated from bank zero when the program starts to execute. Dynamic memory can also be allocated at run time by using the library functions calloc and malloc. All functions and libraries are placed in a static code segment; this segment is called the blank segment, and has no name. With the small memory model, global scalars, arrays and structures are also placed in the blank segment. If the large memory model is used, global scalars are placed in a separate segment called ~GLOBALS, and global arrays and structures are placed in a third segment called ~ARRAYS. Like the blank segment, ~GLOBALS and ~ARRAYS are static segments.

The blank segment and the ~GLOBALS segment are each limited to 64K bytes in length; only the ~ARRAYS segment can be larger than 64K bytes. If the program is larger than 64K bytes in length, the small memory model cannot be used with a single segment. When this happens, there are two alternatives. If the program has many large arrays, the memorymodel directive, described in Chapter 12, can be used to create the ~GLOBALS and ~ARRAYS segments. This causes the compiler to generate larger, slower code to access the arrays, however. If the program does not use a large number of arrays or structures, but has several thousand lines of C code, the problem may be that the code itself exceeds 64K bytes. Using the large memory model will not help in that case. What is needed is a way to cause some of the functions to be placed in a separate static segment. The segment statement does just that: the operand is a string constant that becomes the name of a new segment. All functions defined after the segment statement are placed in the new segment.

```
segment "parser" /* place subroutines in the parser segment */
```

You can create as many static segments as you like. There are advantages and disadvantages to using a large number of segments. On computers with small amounts of memory, or where other programs have fragmented memory, small segments are more likely to load, since there is a better chance that a piece of memory large enough to hold the segment will be found by the loader. A program made up of several segments, however, creates a large number of inter-segment references in the relocation dictionary. These relocation records take up room on the disk, and slow down the loader.

The functions that make up a particular segment do not have to appear in the same source file, nor do they have to appear next to each other in a source file. During the link process, the linker combines all of the functions that have the same segment name into the same load segment, regardless of the order they appear in the source program.

After using the segment statement, if you wish to place a function in the blank segment, code a segment name with ten spaces.

The segment statement can also be used to create dynamic segments. A dynamic segment is not loaded until a call is made to one of the functions in the segment. Once a call is made to a function in the dynamic segment, the segment remains in memory until an explicit call is made to the loader to unload the segment. If more than one segment statement is used to create a dynamic segment, each of the segment statements must specify that the segment is dynamic. If a single function that is not dynamic appears in the load segment, the entire load segment will become a static load segment.

```
segment "initial", dynamic
```

If you have a choice between using the large memory model or the segment statement (as might be the case in a program with several large arrays and a lot of executable code, it is best to use the segment directive, rather than the large memory model, since the compiler generates smaller, faster code when the small memory model is used.

Asm Statement

```
asm-statement:  asm '{' {asm-line}* '}' ';'
asm-line:  {identifier ':'} op-code {operand} {';' comment}
```

The asm statement allows you to code assembly language statements in the body of a function. The syntax for these statement varies a bit from the syntax used by assemblers; these differences are due to the C language itself.

Each of the assembly language statements consists of an operation code. The operation codes are the standard three-letter 65816 operation code mnemonics found in standard references for the 65816. The operation code can be specified in uppercase, lowercase, or a mix of cases. They will not be described in this manual.

Each operation code may be preceded by a label. Unlike assemblers, this label is formatted like a C label. The name of the label follows the same rules as any C identifier, and is case sensitive. Like labels in C, it must be followed by a colon. The label may be used to identify branch points, data locations, or statements.

Many 65816 operations require an operand. This operand is coded after the operation code. It follows the same syntax as is used by assemblers, with a few minor exceptions. First, spaces are allowed in the operand. Second, while expressions may be used, they are limited in form. Expressions must be constant expressions or global or local variables. Global and local variable names may be followed by a + or - operator and a constant expression. Global labels are treated as absolute addresses if the small memory model is in use, and long addresses if the large memory model is in use, unless the addressing mode is modified. (Modification of addressing modes is covered below.) Local variables are treated as direct page locations. If the size of all of the local variables is larger than 255 bytes, this can cause errors in some cases. The mini-assembler does not detect this error. For that reason, it is up to the programmer to ensure that the size of local variables is kept to a minimum if they will be used as the operands of assembly language statements.

The constant expressions used in operands follow the rules for constant expressions in C. All operators and constant operands that can appear in a C constant expression are also allowed in the operand field of an assembly language statement. In particular, note that integer constants use the C syntax for hexadecimal and octal numbers, not the syntax found in most books on assembly language.

When an expression is used in any of the operands, you can force the value to be one byte (forcing direct page addressing), two bytes (forcing absolute addressing) or three bytes (forcing long addressing). This will override the default addressing mode. To force a particular addressing mode, precede the expression with one of the characters shown in the table below.

character	addressing mode
<	direct page
|	absolute
>	long absolute

The syntax for the various addressing modes is shown in the table below. For descriptions on what the addressing modes do, see any of the reference books on the 65816. For addressing modes that have the same physical format, the actual addressing mode used depends on the value of the expression and the kinds of labels used, as described earlier. For example, an operand value of 255 would trigger direct page addressing, 1000 would trigger absolute addressing, and 100000 would trigger absolute long addressing. When a register is shown, such as the A in the accumulator addressing mode, it is shown in uppercase, but may be coded in uppercase or lowercase in a program.

There is one restriction on relative branch operands that does not apply to the other instructions. Relative branches must be made to a label; you cannot code a constant value, nor can you code a label plus or minus some offset.

addressing mode	format
absolute	expression
absolute indexed by X	expression,X
absolute indexed by Y	expression,Y
absolute indexed indirect	(expression,X)
absolute indirect	(expression)
absolute indirect long	[expression]
absolute long	expression
absolute long indexed by X	expression,X
accumulator	A
block move	expression,expression
direct page	expression
direct page indexed by X	expression,X
direct page indexed by Y	expression,Y
direct page indirect	(expression)

direct page indexed indirect	(expression,X)
direct page indirect long	[expression]
direct page indirect indexed	(expression),Y
direct page indirect long indexed	[expression],Y
immediate	#expression
implied	(no operand)
relative addressing	expression
stack relative	expression,S
stack relative indirect indexed	(expression,S),Y

The operand, or operation code if there is no operand, may be followed by a semicolon. If so, this signals the start of an assembly language comment, and all characters from the semicolon to the end of the line are ignored.

Preprocessor macros are still processed within the assembly language statements, and preprocessor macros can be used in the asm statement. The syntactic rules for using the preprocessor and macros in the assembly statement are exactly the same as they are in any other location in the C program. C style comments can also appear the the asm statement.

There are three directives supported by the built-in assembler. They are dcb, dcw, and dcl, which create a byte, word or long word (long int) variable, respectively. The operand for each of these directives is an expression which is evaluated and used to initialize the space.

The short example shown below loads a C integer value i, counts the bits that are set, and stores the result in the C integer variable j.

```
asm {
          lda    i
          ldx    #0
          ldy    #16
   lb1:   lsr    a
          bcc    lb2
          inx
   lb2:   dey
          bne    lb1
          stx    j
   }
```

Chapter 19 – Libraries

Overview of the Libraries

ORCA/C comes with a powerful set of library functions. Most of these are standard libraries that are provided in any good C compiler. A few are unique to ORCA/C or to implementations of C on Apple computers. These functions have been added to make it easier to deal with the Apple IIGS toolbox.

The standard C libraries are listed in alphabetical order for easy reference. This layout is not the best for learning to use the libraries, but it makes it easy to look up a library function to see exactly how it is implemented in ORCA/C. If you are new to C, the book that you are using to learn C should give a more tutorial introduction to the libraries.

The Apple IIGS Toolbox

In addition to the standard C libraries described in the next section, ORCA/C comes with a complete set of tool interface files for the Apple IIGS toolbox. These libraries have been licensed from Apple Computer, inferring that ORCA/C is compatible with source code written for other compilers on the Apple IIGS. These libraries are organized as one interface file per tool. Because the names of the tools, and in fact the tools that are available, are constantly changing on the Apple IIGS, we will not list the library names here. Instead, catalog the folder ORCACDefs in the library folder to find the names of the current tool interface files. These interface files are ASCII files, so you can edit, print, or even change them, if necessary.

System Functions

The run-time library for ORCA/C contains a number of functions that can be manipulated directly from ORCA/C. In some cases, these are functions normally called by the C startup code, or for some other internal purpose, that perform some service you may need in an unusual circumstance. In these cases, you can declare the function as extern and call it from within your C program. In other cases, the default action may not be what you want; for example, you may want to intercept run-time errors, displaying them in a dialog or trapping them for internal handling. In these situations, you can define the function in your program, and ORCA/C will use your version rather than the one from the library.

Source code for all of these subroutines can be found in ORCA Sublib Source, a separate package containing the source code for the ORCA subroutine libraries.

SysCharErrout

```
extern pascal void SysCharErrout (char);
```

Writes a character to error out.

SysCharOut

```
extern pascal void SysCharOut (char);
```

Writes a character to standard out.

SysIOShutDown

```
extern pascal void SysIOShutDown (void);
```

Closes any files opened by SysIOStartup.

SysIOStartup

```
extern pascal void SysIOStartup (void);
```

Starts the I/O system using the files in prefixes 10, 11 and 12. If the files have already been opened, the existing open file is used, and the file is not closed when the program exits. If prefix 10 is .CONSOLE, `DRead` calls are used to read lines of text, allowing editing. In this case, you can use RETURN to finish a line, or CTRL@ or command. to signal an end of file.

SysKeyAvail

```
extern pascal int SysKeyAvail (void);
```

Returns 1 (TRUE) if there is an input character available, and 0 (FALSE) if there is not.

If a character has been put back with a call to `SysPutback`, and has not been read by a subsequent call to `SysKeyin`, the result is TRUE. The rest of the discussion assumes there is no character in the putback buffer.

If input is redirected from a file, this function is equivalent to a test for end of file, returning the opposite result.

For input from .CONSOLE, if there is remaining input in the line buffer, TRUE is returned. If not, and if the Event Manager is active, the result is TRUE if there is a keypress or auto key event available, and FALSE if not. If the Event Manager is not active, the result is TRUE if bit 7 of 0x0C0000 is set, and FALSE if not. (0x0C0000 is the hardware keyboard input location.)

Note: If input is from .CONSOLE, the fact that this function returns TRUE is *not* a guarantee that a call to `SysKeyin` will return immediately, since `SysKeyin` would wait for an entire line to be typed.

SysKeyin

```
extern pascal char SysKeyin (void);
```

Reads a character from standard in. If an end of file condition occurs, (char) 0 is returned.

If input is from .CONSOLE, an entire line is read on the first call to this subroutine, and remaining characters are returned on subsequent calls until the line is exhausted; another call will then read in a new line.

SysPutback

```
extern pascal void SysPutback (char);
```

Places a character in a one-character putback buffer. This character will be the next character returned by `SysKeyin`, and `SysKeyAvail` will return TRUE until the buffer is emptied.

If another call is made to `SysPutback` before the first character is used, the original character is lost.

SystemEnvironmentInit

```
extern pascal void SystemEnvironmentInit (void);
```

This subroutine initializes global variables used by the compilers and their libraries. It should be called by programs that are not started in the normal way as one step in initializing the run-time environment.

SystemError

```
extern pascal void SystemError (int);
```

When a run-time error occurs, libraries call `SystemError`. By defining your own version of `SystemError`, you can intercept and handle run-time errors within your own program. You can also call `SystemError` from within your own program if your own program needs to report an error.

By default, `SystemError` calls two other library subroutines, `SystemPrintError` and `SystemErrorLocation`, to actually handle the error. If you write your own version of `SystemError`, you may want to call one or both of these subroutines for some or all of the errors.

The table below shows the various error numbers currently reported by the run-time libraries. Some of these errors are used by only one of the ORCA languages, so not all of them are actually possible from within a program written entirely in C.

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

SystemErrorLocation

```
extern pascal void SystemErrorLocation (void);
```

This subroutine is called by `SystemError` when a run-time error is reported. Normally, this subroutine prints any traceback information recorded due to the debug pragma, then shuts down the program. This subroutine can be called from within a C program to print traceback information during the debug cycle, or replaced with a different subroutine that either handles an error and recovers from it, or shuts down the system in a different way.

SystemMinStack

```
extern pascal void SystemMinStack (void);
```

This subroutine finds the start of the segment containing the return address, setting the variable `~MinStack` to this value. It should be the very first subroutine called by programs that are not started in the normal way, assuming the program owns the stack frame. `~MinStack` must be set before calling `SystemSANEInit` or before using any debug options that check for stack overflows. It can be set manually from assembly language.

SystemMMShutDown

```
extern pascal void SystemMMShutDown (void);
```

This subroutine shuts down the memory manager used by the run time libraries. It should be called just before a program exits for the last time. The memory manager is left in a restartable state after this call.

SystemPrintError

```
extern pascal void SystemPrintError (int);
```

Writes a text error message to standard out. See `SystemError` for a list of the errors that `SystemPrintError` can handle, as well as the strings it will print.

SystemQuitFlags

```
extern pascal void SystemQuitFlags (unsigned);
```

This subroutine sets the quit flags field for the GS/OS Quit call that is made to exit from a normal C program. See Apple IIGS GS/OS Reference for the allowed values for this parameter.

Note: In restartable programs, be sure to initialize this variable to 0 manually. The libraries do not normally initialize this value.

SystemQuitPath

```
extern pascal void SystemQuitPath (GSString255Ptr);
```

This subroutine sets the quit pathname field for the GS/OS Quit call that is made to exit from a normal C program. See Apple IIGS GS/OS Reference for the allowed values for the parameter.

Note: In restartable programs, be sure to initialize this variable to NULL manually. The libraries do not normally initialize this value.

SystemSANEInit

```
extern pascal void SystemSANEInit (void);
```

This subroutine is called to start SANE. Replacing it with a dummy subroutine would cause the system to skip starting SANE. Calling this subroutine from a CDA or an NDA is a quick way to start SANE, which is not normally started by ORCA/C for these kinds of programs.

This subroutine keeps track of whether SANE was initially started, starting SANE only if needed. `SystemSANEShutDown` will only shut down SANE if it was started by this subroutine.

SystemSANEShutDown

```
extern pascal void SystemSANEShutDown (void);
```

If SANE was started by an earlier call to `SystemSANEInit`, this subroutine shuts down the tool.

SystemUserID

```
extern pascal void SystemUserID (unsigned, char *);
```

This subroutine should be called right after `SystemMinStack` by programs that are not started in the normal way. The first parameter must be passed; it is the user ID for the program. The second parameter can either be a pointer to a command line string or NULL. If the string is a pointer to a command line string, the command line should start with an 8 character identifier naming the launcher, and be followed by the command line as a null terminated string.

Standard C Libraries

abort

See exit.

abs labs fabs

```
#include <stdlib.h>
int      abs(int x);
long     labs(long x);

#include <math.h>
extended fabs(extended x);
```

The function abs accepts an integer argument and returns the absolute value of the argument. Note that abs is in stdlib.h, not math.h, as with some older C compilers.

The function labs accepts a long integer argument and returns the absolute value of the argument. Note that labs is in stdlib.h, not math.h, as with some older C compilers.

The function fabs accepts an extended floating-point argument and returns the absolute value of the argument.

```
distance = fabs(x1-x2);
```

acos

```
#include <math.h>
extended acos(extended x);
```

The function acos returns the trigonometric arc cosine (inverse cosine) of the argument. The result is in radians, and lies in the range 0 to π. If the argument is less than -1.0 or greater than 1.0, errno is set to EDOM.

```
    angle = acos(arg);
```

asctime

See ctime.

assert

```
#include <assert.h>
void assert(int v);
```

Assert is a macro generally used while a program is under development. It takes a single argument, which must be an expression that would be legal as the condition expression in an if statement. If that argument is zero, assert prints "Assertion failed: file file, line number" to standard out, where file is the name of the source file, and number is the line number within the source file. The program is then stopped by calling exit(-1). If the #pragma debug directive has been used to enable trace backs, ORCA/C will also print the line number and source file name where the assert call was made, and a trace back showing what calls were made to arrive at that point.

The macro NDEBUG is used to disable the debug code generated by calls to assert. If NDEBUG is defined when the assert.h header file is read, no code is generated for the assert calls.

```
    assert(parm != 0.0); /* we should never be passed a value of zero! */
```

asin

```
#include <math.h>
extended asin(extended x);
```

The asin function returns the trigonometric arc sine (inverse sine) of the argument. The result is in radians, and lies in the range $-\pi/2$ to $\pi/2$. If the argument is less than -1.0 or greater than 1.0, errno is set to EDOM.

```
    angle = asin(arg);
```

atan

```
#include <math.h>
extended atan(extended x);
```

The atan function returns the trigonometric arc tangent (inverse tangent) of the argument. The result is in radians, and lies in the range -π/2 to π/2.

In some versions of C, this function is called arctan. ORCA/C includes arctan as a macro equivalent of atan to make it easier to port programs written under these compilers.

```
   angle = atan(arg);
```

atan2

```
#include <math.h>
extended atan2(extended y, extended x);
```

The atan2 function returns the trigonometric arc tangent (inverse tangent) of the arguments. Since the actual coordinates of a point are given, rather than the quotient of the two points (as with atan), this function can return results in the range -π to π. The result is in radians, and represents the angle between the positive x axis and the point (x, y) in Cartesian coordinates. A domain error occurs if both x and y are zero, in which case errno is set to EDOM.

```
   angle = atan2(y,x);
```

atexit

```
#include <stdlib.h>
int atexit(void (*func)());
```

The atexit function registers a function so it will be called when the program is complete, either due to a call to the exit function, or because a return is made from the main function. More than one function can be registered in this way; if so, they are called in reverse of the order in which atexit was called to register the functions. The functions must not require parameters, and should return void. The atexit function returns a non-zero value if the function is registered successfully, and zero if there is not enough memory to satisfy the request.

If the same function is registered more than once, ORCA/C will call the function once for each time it is registered. Other compilers may handle this situation differently.

```
   void hello(void)

   {
   printf("Hello, world.\n");
   }
   int main(void)

   {
   atexit(hello);
   }
```

atof atoi atol

```
#include <stdlib.h>
double atof(char *str);
int    atoi(char *str);
long   atol(char *str);
```

These functions are simpler versions of the string conversion functions strtod and strtol. The definitions below define these functions in terms of their more powerful counterparts. For details on what strings are accepted and how errors are handled, see the description of the strtod function.

```
   double atof(char *str)

   {
   return strtod(str, (char**)NULL);
   }

   int atoi(char *str)

   {
   return (int) strtol(str, (char**)NULL, 10);
   }

   long atol(char *str)

   {
   return strtol(str, (char**)NULL, 10);
   }
```

bsearch

```
#include <stdlib.h>
void *bsearch(void *key, void *base, size_t count, size_t size,
              int (*compar)(const void *ptr1, const void *str2));
```

The bsearch function performs a binary search. It searches the array pointed to by base. This array consists of count elements, each of which is size bytes long. The array must be sorted in ascending order. The parameter key points to a value of the same type as the elements of the array; key is the element to search for. The function compar is supplied by the program; it takes two arguments whose types match the type of key, and returns 0 of the arguments match, -1 of the first argument is less than the second, and 1 if the first argument is greater than the second. If the value is found, a pointer to the array element is returned; otherwise, bsearch returns NULL.

```
int CompareZip(address *addr1, address *addr2)

{
if (addr1.zip == addr2.zip)
   return 0;
if (addr1.zip < addr2.zip)
   return -1;
return 1;
}

...
/* find an address with a zip code of 87114 */
addr.zip = 87114;
aPtr = bsearch(&addr, addressList, listSize, sizeof(address),
   CompareZip);
```

c2pstr p2cstr

```
#include <string.h>
char *c2pstr(char *string);
char *p2cstr(char *string);
```

These functions are used to translate between null-terminated C strings and the so-called Pascal strings, which have a leading length byte. In both cases, a pointer to the resulting string is returned. The original string is not changed in any way; the result string is built in an internal buffer. Because an internal buffer is used, subsequent calls to either of these functions can destroy old values. For that reason, it is important to copy the result to a local string buffer if one of these functions will be called before the need for the converted string has passed.

The function c2pstr converts a null-terminated string into a string with a leading length byte. If the null-terminated string is longer than 255 characters, a string with 255 characters is created. The string has a terminating null character following the last character, so standard C string manipulation functions can still be used on the result.

The function p2cstr converts a string with a leading length byte into a null-terminated string.

These functions are not standard C functions. They are included in Apple IIGS based C compilers to make it easier to deal with the toolbox, which often requires strings with a length byte.

```
/* use a null-terminated string to set a window title */
SetWTitle(strcpy(title, c2pstr(temp)), window);
```

calloc

See malloc.

ceil floor

```
#include <math.h>
extended ceil(extended x);
extended floor(extended x);
```

The function ceil accepts a floating-point argument and returns the floating-point representation of the argument, rounded up to the next higher integer. If the argument is an integer, the result is the same value.

The function floor is similar, except that the result is rounded down towards negative infinity.

```
    x = ceil(x);
```

cfree

See free.

chmod

```
#include <fcntl.h>
int chmod(char *path, int mode);
```

Changes the access bits in the file. The following bit flags are supported.

0x0100	Read	enables the file for input
0x0080	Write	enables the file for output
0x1000	Delete	allows the file to be deleted
0x2000	Rename	allows the file to be renamed
0x4000	Backup	indicates that the file should be backed up
0x8000	Invisible	makes the file invisible to the Finder

The flags are added together to make up the mode field. Setting a flag enables the corresponding GS/OS bit, enabling the action described. Clearing the flags disables the action. For example, the call

```
    chmod("myfile", 0x6100);
```

sets the access bits so that the file "myfile" can be read or renamed, and indicates that it should be backed up. The file cannot be written to or deleted without changing the access bits. The file is visible to the Finder.

The flags for Delete, Rename, Backup and Invisible are unique to the Apple IIGS. All of the bits in 0x0E7F are used for other purposes under UNIX. They are ignored in ORCA/C.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible. In particular, the function should not be used in programs that will be ported to other computers unless it is imbedded in conditional compilation code so that the function will only be called on the Apple IIGS version of the program.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

If the file does not exist, errno is set to ENOENT.

clalloc

See malloc.

clearerr

See ferror.

clock

```
#include <time.h>
#define CLK_TCK (60)
typedef unsigned long clock_t;

clock_t clock(void);
```

The clock function is used on multi-tasking systems to see how much time has been used by a program. The result is returned as the number of clock ticks since the program started. The number of clock ticks per second is defined by the macro CLK_TCK; on most systems, this time is returned in microseconds. (One microsecond is 10^{-6} seconds.)

The Apple IIGS operating system is not a multi-tasking operating system, and the Apple IIGS clock is not accurate to the microsecond time scale. ORCA/C uses the tick count returned by the Miscellaneous Tool Set GetTick call for a clock count; CLK_TCK is therefore 60. The tick count is started whenever a heartbeat interrupt handler is installed. The Event Manager installs a heartbeat interrupt handler, so the clock function can always be used from the desktop development environment. If you will be using the clock function from the text environment, you will need to ensure that the heartbeat interrupt handler is active.

```
   clicks = clock();
```

close

```
#include <fcntl.h>
int close(int filds);
```

The file with the file ID filds is closed. If the file has been duplicated using the dup call, it is not closed until each of the associated file IDs have been closed.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set to EBADF (filds is not a valid file descriptor).

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fclose.

commandline

```
#include <orca.h>

char *commandline(void);
```

The commandline function returns a pointer to a null-terminated string. The string represents the command line used to invoke the program, with any input and output redirection removed. Not all shells provide the command line. For those that do not, this function returns NULL.

This function is unique to ORCA/C. In general, the argc, argv functions should be used. This function does have one advantage over argc and argv, though: the line is unparsed, so the program can do its own parsing.

```
   printf("command line: %s\n", commandline());
```

cos

```
#include <math.h>
extended cos(extended x);
```

The cos function returns the trigonometric cosine of the argument. The argument must be supplied in radians.

```
   length = cos(x)*hypotenuse;
```

cosh

```
#include <math.h>
extended cosh(extended x);
```

The cosh function returns the hyperbolic cosine of the argument. If an error occurs, errno is set to ERANGE.

```
   n = cosh(x);
```

creat

```
#include <fcntl.h>
int creat(char *path, int mode);
```

Creates a new file or opens an existing one for output. If the file exists, its length is set to 0. The name of the file is path. The mode parameter is identical to the mode parameter for the chmod call. The file created is a binary file. If the file already exists, it's file type is not changed.

Please note that in APW C, creat does not have a mode parameter. It does in UNIX based C implementations, so we have maintained that use here.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

Possible errors

EACCES	The file exists, and is not write enabled.
EACCES	The file does not exist, and could not be opened.
ENOENT	The pathname is null.
EMFILE	OPEN_MAX files are already open.

ctime asctime

```
#include <time.h>
typedef unsigned long time_t;

char *ctime(time_t *timeptr);
char *asctime(struct tm *ts);
```

The ctime function takes a pointer to an encoded time as input and returns a pointer to an ASCII string of the form

```
Www Mmm dd hh:mm:ss 19yy\n\0
```

where the fields are:

Field	Example	Description
Www	Mon	day of week
Mmm	Feb	month
dd	29	date
hh	16	hour (24 hour format)
mm	03	minutes
ss	57	seconds
yy	88	year

The asctime function creates a similar time string, but takes a pointer to a calendar time structure created by localtime or gmtime as input.

The return string is in a static buffer which is reused by each call to ctime or asctime. If you must keep a copy of the string, and subsequent calls will be made to ctime, be sure to save a copy of the string in a local buffer.

See also time, gmtime, localtime.

```
int main (void)

{
time_t bintime;
struct tm timestruct;

bintime = time(NULL);
printf("The time is %s\n", ctime(&bintime));

timestruct = *gmtime(&bintime);
printf("The time is %s\n", asctime(&timestruct));
}
```

difftime

```
#include <time.h>
typedef unsigned long time_t;

double difftime(time_t t1, time_t t0);
```

The difftime function returns the difference between t0 and t1, in seconds. The parameters are specified in the format used by the time function.

```
printf("%.0f seconds have elapsed.\n", difftime(time(NULL),
   oldtime));
```

div ldiv

```
#include <stdlib.h>
typedef struct div_t {int quot,rem;};
typedef struct ldiv_t {long quot,rem;};

div_t  div(int n, int d);
ldiv_t ldiv(long n, long d);
```

The function div computes the result of division and the remainder as a single step. The quot (quotient, or result of division) and rem (remainder) are returned in a structure. The function ldiv does the same thing for long arguments. The results are unpredictable if the denominator is zero.

```
/* print a result as a whole number and fraction */
res = div(numerator, denominator);
printf("%d and %d/%d", res.quot, res.rem, denominator);
```

dup

```
#include <fcntl.h>
int dup(int old);
```

Duplicates a file ID. The original file ID is created by a creat or open call. This call creates a duplicate of the file ID. The actual file on disk is not closed until each individual file ID is closed.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

Possible errors

EBADF	Old is not a valid file descriptor
EMFILE	OPEN_MAX files are already open

enddesk

See startdesk.

endgraph

See startgraph.

EOF

```
#include <stdio.h>
#define EOF (-1)
```

EOF is a value used to see if you have reached the end of a file. If so, the file input routines will report this character as the one read. After seeing this character, always use the feof function to ensure that the end of the file has, indeed, been reached, as opposed to the file containing a character 0xFF which has been sign extended to become a -1.

```
/* process the characters in a file */
do {
   done = (ch = fgetc(myFile)) == EOF;
   if (done)
      done = feof(myFile);
   if (!done)
      process(ch);
   }
while (!done)
```

errno perror strerror

```
#include <errno.h>
extern int errno;

#include <string.h>
char *strerror(int errnum);

#include <stdio.h>
void perror(char *s);
```

This collection of functions, variables and macros implement the standard run-time error package for C.

The variable errno is used by the libraries to report errors. Any time an error is detected by one of the libraries, an error number is stored in errno. Note that errno is never cleared by the libraries. To make effective use of errno, your program should clear errno before calling a function, call the function, then check errno to see if an error occurred.

The error numbers used by errno are defined as macros in two interface files. Mathematical errors can be found in math.h, while all other errors are defined in errno.h.

The perror function is used to print an error message. The error message is printed to error out. It consists of an error message (which is supplied as a parameter to perror), a colon and a space, a description of the error currently reported by errno, and a new line character.

The strerror function returns a pointer to a string. If errno is supplied as the argument, the result is identical to the message description printed by perror. (This is not true in all implementations of C, however.)

See also toolerror.

```
if (errno) {
   perror("Error at line __LINE__ of __FILE__");
   exit(errno);
   }
```

exit _exit abort

```
#include <stdlib.h>
void exit(int status);
void _exit(int status);
void abort(void);
```

The exit function exits the program, calling all functions registered by the function atexit, and then performing the normal clean-up operations of closing open streams, flushing buffers, and deleting files created by calls to tmpfile. The _exit function also exits, but does not do the clean-up operations. Any open streams will still be closed by the shell, which will also dispose of any memory used by the program. Both functions accept an integer argument, which is returned to the shell as a completion code. A completion code of zero tells the shell that the program finished normally, and any script files will continue to execute. A non-zero value tells the shell that some error occurred. Some shell utilities also use the return code to return a value to the shell.

The abort function is the same as _exit(-1). Abort is not always implemented the same way. Depending on the implementation, abort can exit with some other exit code (so long as it is non-zero), or even exit with a maskable interrupt, so that the program can handle the situation as an error and return to the caller, often returning a value.

```
exit(0);
_exit(errno);
```

exp

```
#include <math.h>
extended exp(extended x);
```

The exp function returns the extended floating-point representation of e raised to the x power, where e is the base of the natural logarithm (approximately 2.718281828). If the exponent cannot be represented as an extended number, an overflow results. In that case, infinity is returned and errno is set to ERANGE.

```
res = exp(x);
```

fabs

See abs.

fclose

```
#include <stdio.h>
int fclose(FILE *stream);
```

The function fclose takes an open file as input, and closes the file. If an error occurs, fclose returns EOF; otherwise, it returns zero.

```
fclose(myfile);
```

fcntl

```
#include <fcntl.h>
int fcntl(int filds, int cmd, int arg);
```

The function fcntl provides control over open files. The *filds* parameter is the file ID for a previously opened file. Under UNIX, the cmd parameter can take on a number of values, but on the Apple IIGS, the only value that makes sense is F_DUPFD, which makes a copy of the file descriptor. The number of the file descriptor returned is always greater than or equal to *arg*, and has exactly the same characters as the old file descriptor.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

If an error occurs, fcntl returns -1 and sets errno to one of these values; if an errno does not occur, fcntl returns 0.

Possible errors

EBADF	filds is not a valid file descriptor.
EINVAL	The cmd parameter is not F_DUPFD.
EMFILE	There ore no available file descriptors greater than arg-1.
EMFILE	arg is negative.

feof

```
#include <stdio.h>
int feof(FILE *stream);
```

The feof function checks to see if an end of file condition exists for the specified stream. An end of file condition exists if an attempt has been made to read past the end of a file. Note that an end of file condition does not exists if all of the characters in a file have been read, but no attempt has been made to read another character. A value of zero is returned if an end of file condition does not exist; a non-zero value is returned if an end of file has been detected.

```
/* echo a file */
stream = fopen("myfile", "r");
if (stream != NULL)
   do
      putchar(fgetc(stream));
   while (!feof(stream));
fclose(stream);
```

ferror clearerr

```
#include <stdio.h>
int  ferror(FILE *stream);
void clearerr(FILE *stream);
```

The ferror function returns a non-zero value if a read or write error has occurred on the specified stream. It returns a zero if no error has occurred.

If an error condition exists, it can be cleared by a call to clearerr or by closing the file.

```
if (ferror(myFile)) {
   clearerr(myFile);
   HandleError();
   }
```

fflush

```
#include <stdio.h>
int fflush(FILE *stream);
```

The fflush function takes a stream open for output and flushes any internal buffers, writing them to the destination device. The stream remains open. EOF is returned if there is an error; otherwise, zero is returned.

```
    fflush(myfile);
```

fgetc getc getchar

```
#include <stdio.h>
int fgetc(FILE *stream);
int getc(FILE *stream);
int getchar(void);
```

These functions are used to read characters from a stream. The function fgetc and the macro getc do exactly the same thing: the only difference is that fgetc is implemented as a function, while getc is implemented as a macro. In each case, a character is read from the stream and returned. The current position of the stream is updated after each read. If an error occurs during the read or if a read is attempted after the last character in the file has been read, EOF is returned. In that case, the feof should be used to see if the end of file has actually been reached. Errno can also be checked to see if an error has occurred.

The function getchar works like an fgetc(stdin), reading a character from standard in. If stdin is set to the keyboard (the default), control-@ is used to signal the end of a file. The end of a line is signalled with the return key or enter key. Note that the console input routines buffer the characters in chunks of one line to allow the user to edit the line while it is being typed. The getchar function will not return a character until an entire line has been typed.

```
    ch = fgetc(myFile);
    ch = getchar();
```

fgetpos fsetpos

```
#include <stdio.h>
int fgetpos(FILE *stream, fpos_t *pos);
int fsetpos(FILE *stream, const fpos_t *pos);
```

The fgetpos function records the current position in a file. The information needed to restore the file to its position at the time of the call is recorded in a value of type fpos_t. The fsetpos function is used to restore the file to the position at the time of the fgetpos call.

If either call is successful, the function returns 0; otherwise it returns a non-zero value and sets errno. While the exact values returned vary from compiler to compiler, in ORCA/C, these functions return -1 for a failure, and set errno to _IOERR.

See also fseek and ftell.

fgets gets

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
char *gets(char *s);
```

The function fgets reads a string from a file. Up to n-1 characters are read; they are placed in the character array pointed to by s. A pointer to the first character in s is returned if the read was successful. Input stops if an end of file condition is encountered, a new line character is encountered, or if n-1 characters are read. In all cases, a terminating null character is placed after the last character read. The new line character, if encountered, does become part of the string; it is placed just before the terminating null character.

If an end of file is encountered before any characters are read, or if an error condition occurs during the read, fgets returns a null pointer and the buffer s is left undisturbed.

The function gets reads characters into the buffer s, taking the characters from standard in. Characters are read until an end of line character is encountered. Unlike fgets, gets does not include the end of line mark as part of the string.

```
/* process a file */
do {
   fgets(line, 255, myFile);
   process(line);
   }
while (!feof(myFile));
```

floor

See ceil.

fmod

```
#include <math.h>
extended fmod(extended x, extended y);
```

The function fmod returns the floating-point remainder of x/y. If the result cannot be represented as an extended floating-point number, the result is undefined. This function can be thought of as performing the division, then removing the whole-number portion of the result. If y is zero, x is returned.

See also modf.

```
x = fmod(x,y);
```

fopen freopen

```
#include <stdio.h>
FILE *fopen(char *filename, char *type);
FILE *freopen(char *filename, char *type, FILE *stream);
```

The function fopen opens a file. The name of the file to open is specified as a null-terminated string. It must be acceptable to GS/OS as a file name. Currently, GS/OS supports a ProDOS FST and several drivers. The name supplied can be the name of any of the drivers, a full path name, a partial path name, or a file in the default directory (prefix 0).

The second string, type, is used to determine the characteristics of the file to be opened. It consists of one of three flags, r, w or a. Each of these flags can be followed by a + character, which indicates that a file is to be opened for both input and output. Even if the file is open for both input and output, a call to fseek, rewind or fflush must appear between input calls and output calls. The meaning of the flags is:

flag	meaning
r	Open an existing file for input.
w	Create a new file, or delete the contents of an existing one, for output.
a	Create a new file, or append to an existing file, for output.
r+	Open an existing file for input and output. The first record read/written will be at the start of the file.
w+	Create a new file or delete the contents of an old file, opening it for input and output.
a+	Create a new file, or append to an existing one, opening the file for input and output. The first value written will appear after all old entries in the file; reading without positioning the file mark will result in an end of file condition.

In addition, any of these flags may be followed by the character b. If the file does not exist, and the b flag is used, fopen will create a file with a BIN file type (a binary file); if the b flag is not used, the file type will be TXT (a text file). The other effect of the b flag is to change the way the \n character is handled. The C language uses the \n character to mark the end of a line, but not all computers use a \n character to mark the end of a line in a file. There are a number of conventions, but the most common are to use the \n character at the end of a line (e.g. UNIX), to use the \r character at the end of a line (e.g. Apple II, Macintosh) or to use both characters (many MS-DOS programs do this). If you open a file without the b flag, the ORCA/C libraries do their best to hide this difference between the way lines are marked by C and the way they are marked in a file. The ORCA/C libraries automatically strip all \n characters, replacing them with \r characters on both input and output. If you use the b flag in fopen, the libraries do not change these characters.

If the call to fopen completes without error, fopen returns a pointer to a file record. This pointer can then be supplied as the input to calls that require an open stream. If an error occurs, fopen returns a null pointer and stores an error code into errno.

The function freopen is used to change the file associated with an existing stream. It starts by closing the file passed as a parameter, and then opens a new file using the same file buffer. If the open succeeds, freopen returns the original file buffer. If the open fails, freopen returns a null pointer and sets errno.

GS/OS limits the number of files that can be open at one time to 32767. In practice, you would probably run out of memory (or files) before that many files were open. ORCA/C does not impose any additional restrictions. For programs that will be ported to other machines, use the macro SYS_OPEN to determine the maximum number of files that can be open at one time.

```
/* sample calls to fopen and freopen */
FILE *myfile;

myfile = fopen(fileName, "r");    /* open a file for input */
myfile = fopen("out.cc", "w");    /* prepare to write to out.cc */
myfile = fopen(fileName, "a");    /* place new info at the end */
myfile = fopen(fileName, "wb");   /* open a binary output file */
myfile = fopen(filename, "r+");   /* open a file for input & output */
```

fprintf printf sprintf

```
#include <stdio.h>
int fprintf(FILE *stream, char *format, ...);
int printf(char *format, ...);
int sprintf(char *s, char *format, ...);
```

These functions implement formatted output to a stream. They all process characters the same way; the difference is in where the characters are sent. In the case of fprintf, the characters can be sent to any stream open for output. The printf function writes characters to standard out. The sprintf function sends characters to a string, appending a terminating null character. In all cases, the number of characters written to the output device is returned if no error occurred, and EOF is returned if an error was detected. In the case of sprintf, the terminating null character is not included in the character count.

The format string controls the characters that are sent to the output device. In the simplest case the characters in the format string are simply copied to the output device. The string may, however, have embedded conversion specifiers. These take the form of a % character followed by other information. In most cases, a conversion specifier requires a value to convert and write. In that case, the values are taken from the variable length parameter list that follows the format string. The number and type of parameters expected by the conversion specifiers must match exactly with the number and type of parameters specified, or a crash is likely.

The general format for a conversion specifier is shown below. Most of the fields are optional; when this is true, it is noted in the text. The various fields are specified in the order given, with no intervening white space. Not all of the optional fields have any effect on a particular format specifier. In those cases, the field is still allowed, but is ignored.

%	All conversion specifiers start with the % character.
-, 0, +, space, #	Zero or more of these flag characters follow the % character, in any order. The flag characters are described in detail below.

unsigned int	An optional unsigned decimal constant specifies the minimum field width. This constant must not start with a zero, since the zero would be confused with the format flag. An asterisk can also be used for the minimum field width, in which case an integer parameter is used for the field width. If the number of characters needed to represent the value is greater than the field width, all of the characters are still printed. If the number of characters needed to represent the value is less that the minimum field width, extra characters are added to fill out the field. The character used can be a space or zero, and the extra characters can appear to the left or right of the value. These factors are under control of the flags.
.unsigned int	The precision of the field is specified by a period and an unsigned decimal constant. In general, this represents the number of digits that will be used to represent a numeric value. Like the field width, the precision can be specified as an asterisk, in which case an integer is removed from the parameter list and used as the precision. See the descriptions of the individual conversion specifiers for the specific use of this value.
h, l or L	This optional field is a size specifier. The h specifier indicates a short operand. It is valid with the d, o, u, x and X conversion specifiers, and is ignored for all other conversion specifiers. The l designator indicates that an integer value is long. It is valid with the d, o, u, x and X conversion specifiers, and is ignored for all other conversion specifiers. L is used with double-precision numbers to indicate double numbers. It is valid with the e, E, f, F, g and G conversion specifiers, and is ignored for all other conversion specifiers. The s and L designators are not needed by the formatter; they are included to make the format specifiers used here more compatible with those used by sscanf.
conversion	The conversion specifier tells what type of variable is being formatted. This field is required. Each of the individual conversion operators is described below.

The flag characters modify the normal operation of the format specifier.

-	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 only used when the output is right-justified.

+	When a negative number is formatted, it is always preceded by a minus sign. This flag forces positive numbers to be preceded by a plus sign.
space	When a negative number is formatted, it is always preceded by a minus sign. This flag forces positive values to be preceded by a space, lining up positive numbers with negative numbers of equal length.
#	This flag modifies the standard output format for certain numeric conversions. The specific effects are described with the conversion specifiers.

One of the conversion operations shown below must appear as the last character of any conversion specifier.

d or i

One argument is removed from the parameter list and written as a signed, decimal number. If the l flag is used, the argument should be of type long; otherwise, the value should be of type int.

If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.

The prefix is a - character is the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.

The # flag is not used by the d conversion specifier.

The i format specifier is identical to the d designator. It is included for compatibility with fscanf.

u

One argument is removed from the parameter list and written as an unsigned, decimal number. If the l flag is used, the argument should be of type unsigned long; otherwise, the value should be of type unsigned int.

If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.

The #, + and space flags are not used by the u conversion specifier.

o

One argument is removed from the parameter list and written as an unsigned, octal number. If the l flag is used, the argument should be of type unsigned long; otherwise, the value should be of type unsigned int.

If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.

The + and space flags are not used by the o conversion specifier.

If the # flag is used, the number is preceded with a leading zero.

x or X

One argument is removed from the parameter list and written as an unsigned, hexadecimal number. If the l flag is used, the argument should be of type unsigned long; otherwise, the value should be of type unsigned int.

If there is no precision specified, the sequence of digits created is as short as possible to represent the value. If a precision is specified, and the digit sequence is less than the precision, it is padded on the left with zeros to reach the specified number of digits. If the precision is zero and the value is zero, no value is printed. The default precision is one.

The + and space flags are not used by the x conversion specifier.

If the # flag is used, the number is preceded by a leading 0x (for the x specifier) or 0X (for the X specifier).

The x specifier causes the digits a through f and the x in the leading 0x (if present) to be written in lowercase, while the X specifier writes these as uppercase letters. This is the only difference between the two specifiers.

p

One pointer argument is removed from the parameter list and written as a pointer. The format used to write a pointer is implementation-defined; in ORCA/C, %p is completely equivalent to %lX.

c

One argument is removed from the parameter list and written as a character. The argument should be of type unsigned or int.

If the value is not a valid ASCII character, it is still sent to the output device. What effect this has depends on the device. Unless you are familiar with the output device, and are deliberately using a character for some special effect, you should stick to characters with ordinal values from 0 to 127.

The #, + and space flags are not used by the c conversion specifier.

s

One argument is removed from the parameter list and written as a string. The argument should be of type char *, and should point to a null-terminated string.

If no precision is specified, all characters up to, but not including the terminating null character are written to the output stream. If a precision is specified, the number of characters written will be the smaller of the precision and the length of the string.

The #, + and space flags are not used by the s conversion specifier.

b

One argument is removed from the parameter list and written as a string. The argument should be of type char *, and should point to a string with a length byte in the first character position.

If no precision is specified, all characters except the length byte are written to the output stream. Note that this gives one way of writing a null character to the output stream. If a precision is specified, the number of characters written will be the smaller of the precision and the length of the string, as specified by the length byte.

This conversion specifier is not present in ANSI C.

The #, + and space flags are not used by the s conversion specifier.

n

No characters are written to the output device. One argument is used from the parameter list; it must be of type int *. The number of characters that have been written up to this time by this formatting operation is saved to the specified location.

f

One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended.

If there is no precision specified, a precision of six is assumed. The number produced consists of at least one leading digit, but no more than are needed to represent the whole part of the number. This is followed by a decimal point and any fraction digits. The precision determines how many fraction digits are present.

If the precision is zero, no fraction digits are written. The decimal point is also not written unless the # specifier is used.

If the value cannot be represented accurately in the precision allowed, it is still written. The value is simply rounded to the closest value to the correct one.

The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.

e or E

One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended.

The value written is in exponential format. It consists of a leading digit of 1 through 9 (or 0 if the actual value is 0), followed by a decimal point and any remaining digits. This is followed by an exponent, which is an e (for the e specifier) or E (for the E specifier), followed by a plus or minus sign, and at least two exponent digits. If more than three digits are needed to express the exponent, then as few as possible are printed to represent the value.

The precision specifies how many digits appear after the decimal point. If no precision is given, a precision of six is used. If the precision is zero, the decimal point and digits are omitted unless the # flag is present, in which case the decimal point is printed, but no digits follow the decimal point.

The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.

g or G	One argument is removed from the parameter list and written as a signed, floating-point number. The argument can be of type float, double, comp or extended. The g specifier is used to print a value in either exponential or fraction format, depending on which size is more appropriate. If the exponent for the number is less than -3 or greater than or equal to the precision specified, the e format specifier is used. When this happens, the effect is as if the e specifier was used with a # flag and a precision one less than the precision given for the g specifier. If no precision is specified, a precision of six is assumed; a precision less than or equal to zero is converted to a precision of one. If the exponent value is greater than or equal to -3, and less than or equal to the precision, the number is formed using the rules for the f format specifier, with a precision equal to the specified precision less the value of the exponent, and again assuming that the # flag was used. If the # flag was specified for the g conversion specifier, the resulting number is the one sent to the output stream. If the # specifier is not present, any trailing fractional zeros are stripped from the number. If all of the fractional digits are zero, the decimal point is also removed. The prefix is a - character if the argument is negative. For a positive argument, the prefix is + if a + flag is used, a space if the space flag is used, and there is no prefix if neither flag is used.
%	The % specifier is used to write a % character to the output stream. No arguments are removed from the stack. Note that the minimum field width, the - justification character and the pad characters are still used with the % specifier. The precision specifier and all other flags have no effect.

```
printf("Hello, world.\n");

for (i = 0; i < 10; ++i)
   printf("%d\n", i);

str = "Hello, world.";  /* NOTE: str is defined as char *str */
for (i = 0; i < strlen(*str); ++i)
   printf("%*s\n", i, str);

printf("x          sin(x)     cos(x)\n");
printf("-          ------     ------\n");
for (x = 0.0; x < pi; x += pi/10)
   printf("%10f %10f %10f\n", x, sin(x), cos(x));
```

fputc putc putchar

```
#include <stdio.h>
int fputc(char c, FILE *stream);
int putc(char c, FILE *stream);
int putchar(char c);
```

These functions are used to write characters to an output stream. The function fputc and the macro putc do exactly the same thing: the only difference is that fputc is implemented as a function, while putc is implemented as a macro. In each case, the character supplied as a parameter is written to the specified output stream and returned as the value of the function. If an error occurs during the write, EOF is returned instead of the character supplied as a parameter.

The function putchar works like an fputc(c,stdout), writing a character to standard out.

```
   /* write the printing ASCII characters to standard out */
   for (i = 32; i < 128; ++i)
      putchar(i);
```

fputs puts

```
#include <stdio.h>
int fputs(char *s, FILE *stream);
int puts(char *s);
```

The function fputs writes the characters in a null-terminated string to an output stream. The string is not followed by a line feed, although any line feed in the string is written. If the write is successful, zero is returned; otherwise, EOF is returned.

The function puts is similar, but it writes the string to standard out. In addition, puts always writes a line feed character after writing the string.

```
   int main(void)

   {
   puts("Hello, world.");
   }
```

fread

```
#include <stdio.h>
size_t fread(void *ptr, size_t element_size, size_t count,
             FILE *stream);
```

The function fread reads characters from a stream, placing them in a memory location pointed to by ptr. The array should be at least count elements long, with each element element_size bytes long. The value returned is the actual number of elements read, which can be less than count if an error occurs (in which case zero is returned) or an end of file is found (in which case the number of items read before the end of file was encountered is returned). If either count or element_size is zero, no characters are read, and zero is returned.

```
    numItems = fread(array, sizeof(element), 40, myFile);
```

free

```
#include <stdlib.h>
void free(void *ptr);
void cfree(void *ptr);
```

The function free is used to deallocate memory allocated using malloc, calloc or realloc. Once deallocated, the memory can be reused by subsequent calls to malloc or calloc. Memory allocated by malloc is allocated from the toolbox Memory Manager either as a single chunk of memory (for requests over 4K bytes), or as a smaller part of a 4K byte piece of memory. For large chunks of memory, free returns the memory to Apple's Memory Manager, allowing other programs and tools to use the memory. For smaller pieces of memory, free will only return the memory to Apple's Memory Manager if all of the individual pieces have been freed.

Older C libraries require memory allocated with calloc to be freed with a call to cfree, rather than free. In ANSI C, cfree has been deleted, and free is used to deallocate all memory. In ORCA/C, cfree has been retained as a macro that calls free for compatibility with old programs.

See also malloc.

```
    /* deallocate space for the array pointed to by aPtr */
    free(aPtr);
```

freopen

See fopen.

frexp ldexp

```
#include <math.h>
extended frexp(extended x, int *nptr);
extended ldexp(extended x, int n);
```

The function frexp splits a floating-point number into a mantissa and exponent. The mantissa is returned as the result, and the exponent is saved to the integer pointed to by nptr. The mantissa will lie in the range 0.5 inclusive to 1.0 exclusive. The mantissa times 2 raised to the exponent will give the original argument x. If x is zero, n is set to zero and zero is returned.

The function ldexp reverses the effect of frexp. The argument x is multiplied by 2 raised to the power n, and the result is returned. If an error occurs, errno is set to ERANGE.

```
    mantissa = frexp(val, &exponent);
    val = ldexp(mantissa, exponent);
```

fscanf scanf sscanf

```
#include <stdio.h>
int fscanf(FILE *stream, char *format, ...);
int scanf(char *format, ...);
int sscanf(char *s, char *format, ...);
```

These three functions are used to read formatted input. In each case, a format string controls the number of items read, what type they are, and several other characteristics. The only difference between the three functions is the source of the characters to format. The fscanf function reads characters from any stream that is open for input. The scanf function is similar, but always takes its characters from standard in. The sscanf function reads the characters from a null-terminated string.

Each of the functions returns the number of successful scan operations. For example, if the function call is set up to read four integers, and all four are read successfully, the value four is returned. If an input/output error, end of file condition, or improper input data forces the read to stop early, only the number of successful scans is returned. If an end of file is encountered before any assignments are made, the functions return EOF. For the purpose of sscanf, the null terminator at the end of the string is treated as an end of file.

The format string describes how many variables should be read, as well as several other characteristics about the input. It consists of three kinds of characters.

white space	If a white space character is found in the format string, the scanner skips to the next non-white space character. All white space characters are also read from the input stream. The effect, then, is that any white space in the format string causes all white space in the input stream to be skipped.
conversions	Conversion specifiers start with a % character. The syntax for a conversion specifier is quite complex, so its format will be discussed later.
other characters	Any other character must match a character read from the input stream. If the character in the format string does not match the character in the input stream, the scanner stops. The character that caused the mismatch is left in the input stream for the next input call.

There must be exactly one variable in the parameter list for each conversion specifier. (There is an exception to this rule; it will be discussed when the format is explained.) The types of the format specifiers must match the type of the parameter exactly. The parameters are all pointers to a variable location; once a value is read by the scanner, it is stored at the location pointed to by the parameter.

Each conversion specifier has the following fields, in the order listed. Some of the fields are optional; if so, this is mentioned in the text.

%	All conversion specifiers start with the % character.

*	The asterisk is an assignment suppression flag. If present, it tells the scanner that it should read a value from the input stream, but that the value will not be saved. When the assignment suppression flag is used, a pointer is not removed from the variable parameter list.
unsigned int	An optional unsigned decimal constant specifies the maximum field width. If this field is present, no more than the specified number of characters will be read from the input stream. It is possible, however, that fewer than the specified number of characters will be read. The field width must be greater than zero.
h or l	This optional field is a size specifier. When used with an integer variable, h indicates that the variable is short, and l indicates that the variable is long. The l specifier can also be used with the floating-point types, where it indicates a double variable. The h and l specifiers are used with the d, u, o x or X format specifiers. In addition, the l specifier can be used with the f, e, E, g, or G floating-point specifiers. These specifiers are ignored when used with any other format specifier.
conversion	The conversion specifier tells what type of variable is being read. This field is required. Each of the individual conversion operators is described below.

There are a variety of conversion specifiers used to describe the variable to be read. These are shown in the table below.

d	A signed decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used. The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. An optional sign is then allowed (+ or -). Next, the scanner expects zero or more decimal digits. Digits are read until a non-digit character is found, and end-of-file is encountered, or the maximum field width is reached. The digits read are then converted into a signed decimal integer and saved. If there are no digits, the resulting value is zero. The results are not predictable if an overflow occurs.
i	This format specifier is identical to the d format specifier, except that octal and hexadecimal numbers can be read. A number is considered to be octal if the first digit is a zero and the next character is not an x or X. A number is considered to be hexadecimal if the first two characters (after a leading sign) are 0x or 0X. Scanning of an octal number stops when a non-octal digit is found. Scanning of a hexadecimal number stops when a non-hexadecimal digit is found. The rules for converting the characters into a

number are identical to those used by the compiler, but a trailing type marker is not allowed.

u

An unsigned decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used. Except for the fact that a leading sign is not allowed and the numbers are unsigned, this conversion specifier is identical to the d specifier used for signed decimal conversion.

o

An unsigned octal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used. No leading sign is allowed, and only octal digits are scanned, but with those exceptions, scanning works exactly like scanning of a signed decimal integer with a format specifier of d.

While ORCA/C will stop scanning if an 8 or 9 is encountered in the input stream; the way these digits are handled is not consistent across all compilers.

x or X

An unsigned decimal value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be short * if the h size specifier is used, int * if no specifier is used, or long * if the l size specifier is used.

The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. Next, the scanner expects zero or more hexadecimal digits. Digits are read until a non-hexadecimal digit character is found or the maximum field width is reached. The digits read are then converted into an unsigned integer and saved.

A leading 0x or 0X is allowed. If these characters appear, they are skipped; they do count toward the maximum field width.

The x and X conversions are identical. Either conversion specifier will accept uppercase or lowercase hexadecimal digits.

p

A pointer is read from the source stream, converted to the internal format used by the 65816, and saved. Since the exact format of a pointer is implementation-defined, this conversion specifier should only be used to read pointers written with the %p conversion specifier with one of the print commands, like printf.

c

The pointer that is used from the variable list should be of type char *. If no field width is specified, exactly one character is read from the input stream and saved at the specified location. If an end of file condition exists, the conversion fails.

If a field width is specified, the pointer should point to an array of characters that can hold as many characters as the maximum field width. Unless an end of file condition occurs, characters are read and stored in the array until the entire field width has been read. No terminating null character is appended to the end of the array. The conversion operation fails, and no characters are saved, if an end of file is found before the entire field width has been processed.

Note that leading white space characters are not skipped by this conversion.

While a size specifier can be present, it is ignored.

s — One pointer argument of type char * is used. A string is read from the stream and placed at the specified location.

The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. Characters are read until an end of file is found, a white space character is found, or the maximum field width is reached (if one is specified). A null character is added to the end of the characters read. If an end of file is encountered before any non-white space characters have been found, the conversion operation fails.

While a size specifier can be present, it is ignored.

b — The b format specifier works exactly like the s format specifier, except that a string with a leading length byte is saved. The string also has a null terminator. If the input string is longer than 255 characters, the resulting string will be n mod 256 characters long, where n is the actual number of characters read.

Note that this conversion specifier is not present in ANSI C. It is included in ORCA/C to make it easier to deal with p-strings, which are used extensively by the toolbox.

n — No input is read from the stream, and any size or field width specifier is ignored. One argument is used from the parameter list; it must be of type int *. The number of successful conversions that have been performed up to this time by the scan operation is saved to the specified location.

f, e, E, g or G — A signed decimal floating-point value is read from the stream, converted to the internal format used by the 65816, and saved. The type of the pointer for this operation must be float * if no specifier is used, or double * if the l size specifier is used.

The scanner starts by skipping any white space characters in the input stream. These are not counted toward the maximum field width if one is specified. The number itself consists of an optional leading sign, an optional digit sequence, an optional decimal point, another optional digit sequence, and an optional exponent. The exponent, if present, consists of the character e or E, an optional sign, and an optional digit sequence. If no digits appear before the exponent, the resulting value is zero. If no digits appear after the e or E character, the exponent value is zero. If the value is too large to represent, the result is an infinity with an appropriate sign. If the value is too small to represent, the result is zero. Note that the results of an overflow or underflow are not consistent across C compilers; other compilers may not return infinity or zero for overflow and underflow.

All of these format specifiers are identical. They will all accept any form of floating-point value, with or without an exponent.

%	A single % character is expected in the input stream. The field with and size specifiers are ignored if present. No pointer is used from the parameter list.
[...]	A string is read from the input stream. A pointer of type char * is used from the parameter list; the characters read are stored at that location.

This specifier allows you to state exactly what characters are allowed in the input stream. Any characters appearing between the [and] characters are allowed; any other characters are not allowed, and cause the scanner to stop. The characters can be specified in any order, and duplicates are allowed.

If the first character after the [character is the ^ character, the meaning of the characters is reversed. In that case, the characters specified are not allowed in the input stream, and any characters not listed are accepted.

Because of the syntax of the conversion specifier, it is not possible to specify a ^ character as the first character of a list of characters, nor is it possible to use the] character in the list.

Scanning continues until an end of file condition is reached, a character which is not in the allowed set of characters is found (in which case it remains unread), or, if a maximum field width is specified, the maximum number of characters have been read. The characters are stored at the location specified by the pointer in the parameter list, and a terminating null character is added to the characters.

While a size specifier can be present, it is ignored.

```
/* echo a file of integers */
while (fscanf(myFile, "%d", &i))
   printf("%d\n", i);

/* echo white space delimited words from a file */
while (fscanf(myFile, "%50s", str))
   printf("%s\n", str);
```

```
/* read words consisting of characters from a file */
while (fscanf(myFile,
   "%50[abcdefghijklmnopqrstuvwxyz"  /* read a word */
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ]"
   "%*[^abcdefghijklmnopqrstuvwxyz"  /* skip non-word chars */
   "ABCDEFGHIJKLMNOPQRSTUVWXYZ]", str))
   printf("%s\n", str);
```

fseek rewind

```
#include <stdio.h>
int  fseek(FILE *stream, long offset, int wherefrom);
void rewind(FILE *stream);
```

The fseek function is used to position the file pointer in a file. Stream is the file to position. Wherefrom is a code telling how to change the position. Three methods of changing the position are currently defined, and are shown in the table below. In ANSI C compilers, names are assigned for each of these codes; older compilers use the numeric values for the codes. Offset is the number of bytes to move by. If an end of file condition exists when this call is made, it is cleared. If an ungetc has been performed, its effects are ignored. The fseek function returns zero if the call is successful, and non-zero if an error occurs.

code	name	meaning
0	SEEK_SET	Moves to offset bytes from the beginning of the file.
1	SEEK_CUR	Moves offset characters from the current file position. Offset can be a negative quantity, which causes the file mark to move toward the start of the file. If an attempt is made to move before the beginning of the file, the mark is set to the first byte in the file. If an attempt is made to move past the end of the file, random bytes are added to extend the length of the file.
2	SEEK_END	Moves offset bytes relative to the end of the file. As with SEEK_CUR, an attempt to move before the start of the file moves to the start of the file, and an attempt to move past the end of the file extends the file to the necessary length with random bytes.

The rewind function is a simple form of the fseek function. It is equivalent to a call to seek with an offset of zero and wherefrom set to SEEK_SET.

See also ftell.

```
/* read a record based on an index */
fseek(myFile, index*sizeof(record), SEEK_SET);
fread(record, sizeof(record), 1, myFile);
```

fsetpos

See fgetpos.

ftell

```
#include <stdio.h>
long int ftell(FILE *stream);
```

The ftell function returns the current file mark for an open stream. In effect, this is the number of bytes between the start of the file and the next byte that will be read or written. It is generally used to record a position, with the result supplied to fseek at a later time.

If an error occurs, ftell returns -1L and sets errno to the error number of the error detected.

See also fseek.

```
position = ftell(myFile);
```

fwrite

```
#include <stdio.h>
size_t fwrite(void *ptr, size_t element_size, size_t count,
              FILE *stream);
```

The function fwrite writes count elements of an array with elements element_size bytes long to the output stream, taking the values from memory starting at the location indicated by ptr. The function returns the actual number of array elements written, which will be less than count if an output error has occurred.

```
fwrite(matrix, sizeof(float), 10*10, myFile);
```

getc

See fgetc.

getchar

See fgetc.

getenv

```
#include <stdlib.h>
char *getenv(const char *name)
```

The getenv call returns a pointer to the value of a shell variable. If the shell variable has not been set, or if the program is executing from outside of the ORCA programming environment, where shell variables do not exist, a NULL value is returned.

Note that the meaning of getenv varies from environment to environment. On other computers, getenv may return a value from a different source.

gets

See fgets.

gmtime

See localtime.

isalnum

```
#include <ctype.h>
int isalnum(char c);
```

Isalnum is a macro that returns a non-zero value if the argument is an alphanumeric character, and zero if the argument is not an alphanumeric character. The argument must lie in the range -1 to 255, or the result is not valid. The alphanumeric characters are shown below.

```
0 1 2 3 4 5 6 7 8 9
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

for (i = 0; i < 128; ++i)
   if (isalnum(i))
      printf("%c is alphanumeric.\n", i);
```

isalpha

```
#include <ctype.h>
int isalpha(char c);
```

Isalpha is a macro that returns a non-zero value if the argument is an alphabetic character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The alphabetic characters are show below.

```
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   for (i = 0; i < 128; ++i)
      if (isalpha(i))
         printf("%c is alphabetic.\n", i);
```

isascii

```
#include <ctype.h>
int isascii(char c);
```

Isascii is a macro that returns a non-zero value if the argument's ordinal value is in the range 0 to 127, which is the range if the ASCII character set used on the Apple IIGS. It returns zero if the character is outside of the range of the ASCII character set.

```
   for (i = -500; i < 500; ++i)
      if (isascii(i))
         printf("%d is the ordinal value of an ASCII character.\n", i);
```

iscntrl

```
#include <ctype.h>
int iscntrl(char c);
```

Iscntrl is a macro that returns a non-zero value if the argument is a control character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The control characters are all those characters with an ordinal range of 0 to 31, plus the character whose ordinal value is 127.

```
   for (i = 0; i < 128; ++i)
      if (iscntrl(i))
         printf("%d is the ordinal value of a control character.\n", i);
```

iscsym

```
#include <ctype.h>
int iscsym(char c);
```

iscsym is a macro that returns a non-zero value if the argument is a character that can appear in a C identifier, and zero if it cannot. The argument must lie in the range -1 to 255, or the result is not valid. The characters which can appear in a C identifier are shown below.

```
   _
   0 1 2 3 4 5 6 7 8 9
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   for (i = 0; i < 128; ++i)
      if (iscsym(i))
         printf("%c can appear in a C symbol.\n", i);
```

iscsymf

```
#include <ctype.h>
int iscsymf(char c);
```

Iscsymf is a macro that returns a non-zero value if the argument is a character that can appear as the first character of a C identifier, and zero if it cannot. The argument must lie in the range -1 to 255, or the result is not valid. The characters which can appear as the first character of a C identifier are shown below.

```
   _
   a b c d e f g h i j k l m n o p q r s t u v w x y z
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

   for (i = 0; i < 128; ++i)
      if (iscsymf(i))
         printf("%c can appear as the first character in a C symbol.\n",
            i);
```

isdigit

```
#include <ctype.h>
int isdigit(char c);
```

Isdigit is a macro that returns a non-zero value if the argument is one of the ten decimal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The ten decimal digits are shown below.

```
   0 1 2 3 4 5 6 7 8 9

   for (i = 0; i < 128; ++i)
      if (isdigit(i))
         printf("%c is a digit.\n", i);
```

isgraph

```
#include <ctype.h>
int isgraph(char c);
```

Isgraph is a macro that returns a non-zero value if the argument is one of the printing characters, other than a space. The argument must lie in the range -1 to 255, or the result is not valid. See isprint for a list of the printing characters.

```
   for (i = 0; i < 128; ++i)
      if (isgraph(i))
         printf("%c is a graph character.\n", i);
```

islower

```
#include <ctype.h>
int islower(char c);
```

Islower is a macro that returns a non-zero value if the argument is one of the lowercase alphanumeric characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The lowercase alphanumeric characters are shown below.

```
   a b c d e f g h i j k l m n o p q r s t u v w x y z

   for (i = 0; i < 128; ++i)
      if (islower(i))
         printf("%c is a lowercase character.\n", i);
```

isodigit

```
#include <ctype.h>
int isodigit(char c);
```

Isodigit is a macro that returns a non-zero value if the argument is one of the eight octal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The eight octal digits are shown below.

```
   0 1 2 3 4 5 6 7

   for (i = 0; i < 128; ++i)
      if (isodigit(i))
         printf("%c is an octal digit.\n", i);
```

isprint

```
#include <ctype.h>
int isprint(char c);
```

Isprint is a macro that returns a non-zero value if the argument is one of the printing characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The printing characters include the entire ASCII character set with the control characters removed. This includes all characters with an ordinal value of 32 to 126. The printing characters are shown below. Note that the first printing character is a space.

```
     ! " # $ % & ' ( ) * + , - . /
   0 1 2 3 4 5 6 7 8 9 : ; < = > ?
   @ A B C D E F G H I J K L M N O
   P Q R S T U V W X Y Z [ \ ] ^ _
   ` a b c d e f g h i j k l m n o
   p q r s t u v w x y z { | } ~

   for (i = 0; i < 128; ++i)
      if (isprint(i))
         printf("%c is a printing character.\n", i);
```

ispunct

```
#include <ctype.h>
int ispunct(char c);
```

The function ispunct is a macro that returns a non-zero value if the argument is one of the punctuation characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The punctuation characters include the entire ASCII character set with the control characters and alphanumeric characters removed. This includes the space character, plus all of the characters shown below.

```
   ! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~

   for (i = 0; i < 128; ++i)
      if (ispunct(i))
         printf("%c is a punctuation character.\n", i);
```

isspace

```
#include <ctype.h>
int isspace(char c);
```

Isspace is a macro that returns a non-zero value if the argument is a white space character, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The white space characters include the space, tab, carriage return, new line, vertical tab and form feed characters. These are the characters with ordinal values of 9 to 13 and 32.

```
   for (i = 0; i < 128; ++i)
      if (isspace(i))
         printf("%d is the ordinal value of a white space character.\n",
i);
```

isupper

```
#include <ctype.h>
int isupper(char c);
```

Isupper is a macro that returns a non-zero value if the argument is one of the uppercase alphanumeric characters, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The uppercase alphanumeric characters are shown below.

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

isxdigit

```
#include <ctype.h>
int isxdigit(char c);
```

Isxdigit is a macro that returns a non-zero value if the argument is one of the sixteen hexadecimal digits, and zero if it is not. The argument must lie in the range -1 to 255, or the result is not valid. The sixteen hexadecimal digits are shown below.

```
0 1 2 3 4 5 6 7 8 9
a b c d e f
A B C D E F

for (i = 0; i < 128; ++i)
   if (isxdigit(i))
      printf("%c is a hexadecimal digit.\n", i);
```

labs

See abs.

ldexp

See frexp.

ldiv

See div.

localtime gmtime mktime

```
#include <time.h>
typedef unsigned long time_t;
struct tm {
   int tm_sec;    /* seconds; 0 to 59 */
   int tm_min;    /* minutes; 0 to 59 */
   int tm_hour;   /* hours; 0 to 23 */
   int tm_mday;   /* day of the month; 1 to 31 */
   int tm_mon;    /* month; 0 to 11 */
   int tm_year;   /* year; 0 == 1900 */
   int tm_wday;   /* day of week; sun == 0 to sat == 6 */
   int tm_yday;   /* day of year; 0 to 365 */
   int tm_isdst;  /* daylight savings time?; a boolean value */
   }

struct tm *localtime(time_t *t);
struct tm *gmtime(time_t *t);
time_t mktime(struct tm *tmptr);
```

The localtime function takes a time encoded as an unsigned long integer and returns a structure with the various time fields filled in. See the definition of the tm structure, above, for a description of the fields that are filled in. On the Apple IIGS there is no way of telling if the current time is daylight savings time, so localtime always returns false (zero) for that field.

The gmtime function is designed to return the time on the prime meridian, which runs through Greenwich England. This time is known as Greenwich Mean Time, or GMT. Since the Apple IIGS does not have a way of knowing the difference between local time and GMT, the gmtime function returns the same values as localtime in ORCA/C.

The mktime function takes a time structure as input and returns the time in the long integer format returned by the time function. As a side effect, it also recomputes the tm_wday and tm_yday fields based on the values of the other fields. It returns -1 if the call fails.

See also time.

```
   curtime = time(NULL);
   t = *localtime(&curtime);
```

log

```
#include <math.h>
extended log(extended x);
```

The natural logarithm of the argument is returned. If the argument is negative, errno is set to EDOM. If the argument is zero or close to zero, a range error occurs and errno is set to ERANGE.

```
   res = log(x);
```

log10

```
#include <math.h>
extended log10(extended x);
```

The base-10 logarithm of the argument is returned. If the argument is negative, errno is set to EDOM. If the argument is zero or close to zero, a range error occurs and errno is set to ERANGE.

```
    res = log10(x);
```

longjmp

See setjmp.

lseek

```
#include <fcntl.h>
long lseek(int filds, long offset, int whence);
```

Repositions the file pointer, so that the next read or write occurs from a new position. The position is specified by offset. *whence* indicates one of the following:

0	The file pointer is set to offset bytes from the start of the file.
1	The file pointer is set to the current position plus offset bytes.
2	The file pointer is set to offset bytes from the end of the file.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fseek.

<u>Possible errors</u>

EBADF	Old is not a valid file descriptor.
EINVAL	whence is not 0, 1 or 2.

malloc calloc

```
#include <stdlib.h>
void *malloc(size_t size);
void *mlalloc(size_t size);
void *calloc(size_t count, size_t size);
void *clalloc(size_t count, size_t size);
```

The function malloc allocates size bytes of memory and returns a pointer to the first byte of that memory. If there is not enough free memory to satisfy the request, or if the size is zero, malloc returns NULL.

On the Apple IIGS, memory is allocated through Apple's Memory Manager. If you ask for more than 4096 bytes of memory, the memory will be requested from the Memory Manager in a single chunk. There is no guarantee that the memory will not cross a bank boundary. While special memory is only used if all other memory has already been allocated, there is also no guarantee that the memory will not be special memory. (Special memory consists of memory from banks 0, 1, 0xE0, and 0xE1. This memory is used for special purposes on the Apple IIGS.) If a call to malloc is made asking for less than 4096 bytes of memory, 4096 bytes are still allocated from the Memory Manager. This memory is then subdivided by the compiler's run-time memory manager to satisfy other requests for small amounts of memory, taking a burden off of the toolbox's Memory Manager.

The calloc function accepts two parameters, a count and a size. Enough memory is allocated to hold count elements of size size, using the same mechanism for allocating memory as is used by malloc. All of the allocated memory is then set to zero.

Older C libraries often restrict malloc and calloc to a parameter of size unsigned, which would limit these functions to allocating 64K bytes of memory at a time. ANSI C requires parameters of type unsigned long, and eliminates mlalloc, which is used in older compilers when more than 64K bytes of memory is needed. ORCA/C still includes mlalloc as a macro which calls malloc, but its use should be avoided in new programs. The function clalloc was provided for similar reasons; again, clalloc is a macro in ORCA/C; it calls calloc.

See also free, realloc.

```
/* allocate space for the array pointed to by aPtr */
aPtr = malloc(sizeof(*aPtr));
```

memchr

```
#include <string.h>
void *memchr(const void *ptr, int val, size_t len);
```

The function memchr searches for a byte with the value val starting at the location pointed to by ptr for a maximum of len bytes. If a matching byte is found, a pointer to that byte is returned. If a matching byte is not found, a null pointer is returned.

See also strchr. Unlike strchr, memchr does not stop if a zero value is found.

```
   /* skip to the next line in a TXT file */
   file = ((char *) memchr(file, '\r', 255))+1;
```

memcmp

```
#include <string.h>
int memcmp(void *ptr1, void *ptr2, size_t len);
```

The function memcmp compares two areas of memory, one starting at ptr1, and the other starting at ptr2, for a length of len bytes. If all bytes match, memcmp returns zero. If a byte is found that does not match, a negative number is returned if the byte pointed to by ptr1 is less than the byte pointed to by ptr2, and a positive value is returned otherwise.

See also strcmp. Unlike strcmp, memcmp does not stop if null characters are found.

```
   /* see if two files loaded into memory are equal */
   if (memcmp(file1, file2, fileLen) == 0)
      printf("The files are equal.\n");
```

memcpy memmove

```
#include <string.h>
void *memcpy(const void *dest, void *src, size_t len);
void *memmove(void *dest, const void *src, size_t len);
```

The function memcpy copies len bytes from the location starting at src to the location starting at dest. It returns the value of src. If the memory areas overlap, the results are unpredictable. The function memmove is similar, but is guaranteed to work if the memory areas overlap. Regardless of the location of the memory areas, memmove will produce a faithful copy of the original memory area at the destination.

```
   /* initialize a doubly-subscripted array row-by-row */
   for (i = 0; i < maxCol; ++i)
      a[0][i] = 1.0;
   for (i = 1; i < maxRow, ++i)
      memcpy(a[i], a[0], sizeof(a[0]));
```

memmove

See memcpy.

memset

```
#include <string.h>
void *memset(const void *ptr, int val, size_t len);
```

The function memset copies the value val into an area len bytes long, starting at ptr. It returns the value of ptr.

```
/* clear a large array */
memset(array, 0, sizeof(array));
```

mktime

See localtime.

mlalloc

See malloc.

modf

```
#include <math.h>
extended modf(extended x, int *nptr);
```

The function modf returns the fraction part of the argument x. As a side effect, the integer part is placed in the location pointed to by nptr. The fraction part and integer part will have the same sign.

See also fmod.

```
res = modf(x, &i);
```

offsetof

```
#include <stddef.h>
size_t offsetof(type, member);
```

The offsetof macro returns the position of a structure member within a structure. The first input is a struct type. It must be defined in such a way that the expression

```
&(type *)
```

is legal; for example, any type defined like this:

```
typedef struct type {...} type;
```

will work. The second parameter is the name of a variable in the structure. The value returned is the number of bytes that occur before the member in the structure.

open

```
#include <fcntl.h>
int open(char *path, int oflag);
```

Creates a new file or opens an existing one for output.
The *oflag* parameter is formed by oring one or more of the following flags.

O_RDONLY	Open for read only.
O_WRONLY	Open for write only.
O_RDWR	Open for both reading and writing.
O_NDELAY	Not used on the Apple IIGS. Included for UNIX compatibility.
O_APPEND	If this flag is set, the file pointer is advanced to the end of the file before every write operation.
O_CREAT	If the file does not exists, one is created.
O_TRUNC	If the file exists, the length is set to 0 after opening the file.
O_EXCL	If O_EXCL and O_CREAT are both set and the file exists, the call fails.
O_BINARY	The file opened is a binary (BIN) file, rather than the default text (TXT) file. For text files, \n characters are translated to \r characters on output, and \r characters are translated to \n characters on input, conforming to Apple IIGS conventions. For binary files, no such conversion is performed. This flag is unique to the Apple IIGS.

If the call is successful, the function returns 0; otherwise, a -1 is returned and errno is set as indicated in the list below.

Possible errors

EACCES	An I/O error or invalid pathname prevented opening of the file.
EEXIST	O_CREAT and O_EXCL are set, and the file exists.
EMFILE	OPEN_MAX files are already open.
ENOENT	O_CREAT is not set and the file does not exist.
ENOENT	The pathname is null.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fopen.

p2cstr

See c2pstr.

pow

```
#include <math.h>
extended pow(extended x, extended y);
```

The argument x is raised to the power of the argument y, and the result is returned. If x is not zero, but y is zero, 1.0 is returned. If x is zero and y is positive, 0.0 is returned. Domain errors will occur if x is negative and y is not an integer, or if x is zero and y is zero or negative. In these cases, errno is set to EDOM. Range errors can also occur; in these cases, errno is set to ERANGE.

```
cube = pow(num, 3.0);
```

putc

See fputc.

putchar

See fputc.

puts

See fputs.

qsort

```
#include <stdlib.h>
void qsort(void *base, size_t count, size_t size,
           int (*compar)(const void *ptr1, const void *ptr2));
```

The qsort function sorts an array. The array has count elements, each of which is size bytes long. The base parameter points to the first element of the array. The function compar is supplied by the program; it accepts two pointers as arguments, each of which points to an element of the array, and returns an integer as a result. The value returned is 0 if the first element is the same as the second, -1 if the first element is less than the second, and 1 if the first element is greater than the second. After the call, the array will be sorted in ascending order.

```
int CompareZip(address *addr1, address *addr2)

{
if (addr1->zip == addr2.zip)
   return 0;
if (addr1->zip < addr2.zip)
   return -1;
return 1;
}
```

```
   ...
   /* sort the addresses by zip code */
   qsort(addressList, listSize, sizeof(address), CompareZip);
```

raise

```
#include <signal.h>
int raise(int sig);
```

The raise function generates a signal that will be handled by the signal handler. See signal for a way to install a signal handler and a description of signals.

The function returns 0 if the sig parameter is valid, and 1 if it is not. If the signal is not valid, errno is also set to ERANGE.

rand srand

```
#include <stdlib.h>
#define RAND_MAX 32767
int  rand(void);
void srand(int seed);
```

The function srand is used to initialize a pseudo-random number generator. Subsequent calls to rand will return integers in the range 0 to RAND_MAX.

Choosing a seed value is very important. If the same seed is used each time a program is executed, the program will always return the same pseudo-random number sequence. The seed should be chosen in such a way that it will be fairly random itself. One way of choosing a seed that works in most situations is to use the seconds in the current time of day.

```
   time_t t;
   struct tm trec;

   t = time(NULL);
   trec = *localtime(&t);
   srand(trec.tm_sec);
   for (i = 0; i < 100; ++i)
      printf("%8d", rand());
```

read

```
#include <fcntl.h>
int read(int filds, void *buf, int n);
```

Up to n bytes are read from the file whose file ID is *filds*. The bytes are placed in the buffer pointed to by *buf*.

If an error occurs during reading, the function returns -1 and sets errno as indicated in the table below. If the file is already at the end of file mark, a zero is returned, but no error is set. If there are n or more bytes left in the file, n bytes are read, and n is returned; the file pointer is advanced n bytes. If there are less than n bytes left in the file, the number of bytes remaining in the file are read, and this number is returned; the file pointer is then advanced to the end of the file.

If the O_BINARY flag was not set when the file was opened, any \r characters are converted to \n characters as they are read.

Possible errors

EIO	A physical I/O error occurred.
EBADF	filds is not a valid file ID.
EBADF	The file is not open for reading.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fread.

realloc

```
#include <stdlib.h>
void *realloc(char *ptr, size_t size);
void *relalloc(char *ptr, size_t size);
```

The function realloc takes a pointer to a piece of memory allocated previously by malloc or calloc and changes the size of the memory. If the memory area shrinks, the deleted bytes are no longer available, although the memory not deleted is not disturbed. If the memory area grows, the old contents are preserved, but the value of the new bytes is unpredictable. A pointer is returned if the request is successful; the memory area may have been moved, so old copies of the pointer are no longer useful. If the request cannot be satisfied, a null pointer is returned, and the old memory area is not disturbed.

If a null pointer is passed to realloc, the function behaves as if malloc had been called. If the pointer is not null, and the size is zero, the memory is deallocated and NULL is returned.

In older C compilers, realloc was limited to a parameter of size unsigned, and another function called relalloc was used for requests for larger memory. In ANSI C, relalloc does not exist, and realloc can be used to request any amount of memory. ORCA/C includes relalloc as a macro equivalent of realloc for compatibility with older programs.

See also free, malloc.

```
/* Bytes are obtained from InByte, buffered until no   */
/* more memory is available, and processed by DoIt.    */
length = 0;
size = remaining = growSize;
ptr = start = malloc(growSize);
if (ptr == NULL)
   printf("Insufficient memory.\n");
else {
   ptr++ = InByte();
   length++;
   if (--remaining == 0) {
      size += growSize;
      ptr = realloc(start, size);
      if (ptr == NULL) {
         DoIt(start, length);
         free(start);
         length = 0;
         size = remaining = growSize;
         ptr = start = malloc(growSize);
         }
      else {
         start = ptr;
         ptr += length;
         }
      }
   }
```

remove

```
#include <stdio.h>
int remove(char *filename);
```

The named file is deleted from the disk. The file name can be any file name or path name recognized by GS/OS. A zero is returned if the delete is successful, and a non-zero is returned if the delete is not successful.

```
   remove("/hd/languages/apwc");
```

rename

```
#include <stdio.h>
int rename(char *oldname, char *newname);
```

The file with the name represented by the string oldname is renamed. The file names can be any file name or path name recognized by GS/OS. After the rename call, the name of the file is the name given by newname. If the rename is successful, the function returns a zero; otherwise, a non-zero value is returned.

The rename function is implemented by calling the GS/OS rename facility. Because of this, rename is more powerful under ORCA/C than it is under some other C implementations. This call can be used to move a file from one directory to another by specifying the path names of the old and new files.

```
   /* change a n file name */
   rename("myfile", "hisfile");

   /* move a file */
   rename("/hd/system/finder", "/hd/utilities/finder");
```

rewind

See fseek.

scanf

See fscanf.

setbuf setvbuf

```
#include <stdio.h>
void setbuf(FILE *stream, char *buf);
int setvbuf(FILE *stream, char *buf, int type, size_t size);
```

The function setvbuf is used to change the size and characteristics of the default file buffer. By default, when fopen opens a file, the file has a BUFSIZ (1024) byte buffer used to collect characters to avoid the overhead of calls to GS/OS for small read/write requests. The file is fully buffered, which means that, barring intervention with a call to fflush or fclose, 1024 bytes are collected before data is written to the disk, and 1024 bytes are read from disk to satisfy any read request.

When setvbuf is called, the first parameter is a pointer to an open stream. This stream must already be open, but no input or output calls are allowed before the call to setvbuf. The buf parameter is a pointer to a character buffer to use as the file buffer. If buf is a null pointer, setvbuf will allocate a buffer from available memory; if buf is not null, it should be at least size bytes long. Size is the number of bytes to use for a buffer, while type is the type of buffering to use. There are three buffering types, designated using macros defined in the stdio.h interface files. These are:

type	meaning
_IOFBF	Full buffering. A buffer of size bytes is collected before output operations, and a buffer of size bytes is read for any read operation. This is the default buffering mechanism.
_IOLBF	Line buffering. On output, the buffer is flushed any time a new line character is written to the buffer, or when the buffer is full.
_IONBF	No buffering. All output passes directly to GS/OS, and all input is satisfied with direct calls to GS/OS.

The function setbuf is a simplified form of setvbuf. If the buf is a null pointer, no file buffering is used. If the buf pointer is not null, full buffering is used, and a pointer to a buffer of BUFSIZ bytes long is required as the input.

```
/* Use line buffering, using line as a buffer 255 characters */
/* long.                                                      */
setvbuf(myFile, line, _IOLBF, 255);
```

setjmp longjmp

```
#include <setjmp.h>
typedef int jmp_buf[4];

void longjmp(jmp_buf env, int status);
int  setjmp(jmp_buf env);
```

The setjmp and longjmp functions are used to transfer control to a previous point in the program. The setjmp function is used to establish a jump location in the program. A jump buffer is passed; this jump buffer is later used by longjmp. The setjmp function returns a value of zero.

Later, a call is made to longjmp, and the jump buffer and an integer are passed as parameters. The call can be made from the function that called setjmp, or from any function it called or that was called by a function it called (and so on), but longjmp must not be used from a function that called setjmp. Control it returned as if setjmp was call. The call to setjmp is not actually made; instead, a return is simulated. The stack is cleaned up, including disposing of the space used by any local auto variables. The value passed to longjmp as a status, which must be non-zero, is returned as the return value of setjmp. As far as the function that called setjmp in the first place can tell, setjmp has simply returned with a non-zero value.

If longjmp is called with an invalid buffer or the function containing the setjmp call returns before the longjmp call is made, the results are unpredictable.

```
jmp_buf buffer;

void test(void)

{
printf("In test.\n");
longjmp(buffer, 1);
printf("This line will not be executed.\n");
}

int main(void)

{
if (setjmp(buffer))
   printf("Returned from jump.\n");
else {
   printf("Setting up the jump.\n");
   test();
   }
}
```

shellid

```
#include <orca.h>

char *shellid(void);
```

The shellid function returns a pointer to a null-terminated string used by the shell that executed the program to distinguish itself from other shells. Not all shells report a shell ID; for those that do not, this function returns NULL. The shell ID for ORCA and APW is BYTEWRKS.

This function is unique to ORCA/C.

```
printf("shell ID: %s\n", shellid());
```

signal

```
#include <signal.h>
void (*signal(int sig, void(*func)(int)))(int);
```

Signals are used to handle asyncronous interrupts. They are primarily used on multi-tasking systems, where another executing program or some hardware facility can cause an interrupt. In that sort of environment, signals can be used to inform the program what has happened. Signals are not used on the Apple IIGS; the signal.h library is included solely to aid in porting programs from multi-tasking environments.

The signal function is used to specify how the various signals should be handled. The first parameter tells the signal handler which of the various signals you are talking about. ORCA/C supports the standard signals required by the ANSI standard. They are:

SIGABRT	Abnormal termination.
SIGFPE	Arithmetic errors.
SIGILL	Invalid function image.
SIGINT	Interactive attention signal.
SIGSEGV	Invalid memory access.
SIGTERM	Termination request sent to the program.

None of these errors apply to the Apple IIGS, and the ANSI standard does not require any of them to be handled. The SIGILL and SIGSEGV errors simply don't exist on the Apple IIGS. Abnormal terminations on the Apple IIGS are severe enough that attempting to execute a function after one occurs would be foolish; it could damage data on a disk, for example. The other signals either have no direct counterpart on the Apple IIGS, or are handled by the Apple IIGS toolbox, which is not always active.

The second parameter to signal is either SIG_DFL, SIG_IGN, or a function. SIG_DFL tells the signal handler to use default handling, while SIG_IGN tells the signal handler to ignore the signal. With ORCA/C, these are the same, since the default handling is to ignore a signal. If you pass a function, the function will be called whenever the raise function is called to raise a signal. The signal number (e.g., SIGABRT) is passed to your function.

The signal function returns the value of the previous signal handler, which you can save, and then restore later. If the sig parameter is not one of the signals listed above, signal returns SIG_ERR and sets errno to ERANGE.

See raise for a way to create a signal.

sin

```
#include <math.h>
extended sin(extended x);
```

The sin function returns the trigonometric sine of the argument. The argument must be supplied in radians.

```
   height = sin(x)*hypotenuse;
```

sinh

```
#include <math.h>
extended sinh(extended x);
```

The sinh function returns the hyperbolic sine of the argument. If an error occurs, errno is set to ERANGE.

```
   n = sinh(x);
```

sqrt

```
#include <math.h>
extended sqrt(extended x);
```

The square root of the argument is returned. If the argument is negative, errno is set to ERANGE.

```
   length = sqrt(x*x + y*y);
```

srand

See rand.

sscanf

See fscanf.

startdesk enddesk

```
#include <orca.h>

void startdesk(int width);
void enddesk(void);
```

The startdesk function starts the tools shown in the table below. The Tool Locater, Memory Manager, Loader and SANE are started by all C programs. The parameter width should be 320 or 640; it is used to start QuickDraw II, and determines the resolution of the graphics screen. This function eliminates a great deal of the drudgery associated with starting the tools in the Apple IIGS toolbox.

In addition to starting the tools, startdesk also changes the way the standard C libraries handle input and output. If you do not use startdesk (or its cousin, startgraph) any text written to standard out or error out is written to the text screen or the shell window, depending on which programming environment you are using. After using startgraph, though, all text is written using QuickDraw, so it shows up in the current graphport. You can change the location where the text will appear by using the MoveTo call just before you write the text, and you can use the normal QuickDraw and font manager calls to change the font, pen color, size, and style of the text that will be written.

Note that the tool locator, SANE, and the memory manager are started in every C program, so they are not started explicitly by startdesk.

The enddesk function shuts down all of the tools started by startdesk.

Miscellaneous Tools	QuickDraw II	Desk Manager
Event Manager	Integer Math Tool Set	Window Manager
Menu Manager	Control Manager	QuickDraw Auxiliary
Line Edit	Dialog Manager	Scrap Manager
Standard File Manager	Font Manager	List Manager
Resource Manager		

```
startdesk(640);
/* desktop program goes here */
enddesk();
```

startgraph endgraph

```
#include <orca.h>

void startgraph(int width);
void endgraph(void);
```

The startgraph function starts QuickDraw II and initializes it to draw with a white pen on a black background, sets the pen mode to or, and initializes the pen size to 1 pixel by 1 pixel. The parameter should be either 320 or 640; it determines the resolution of the screen. The startgraph function makes it easy to start QuickDraw II, reducing the amount of code you have to write to make a simple graphics program work. Finally, startgraph changes the way the standard C text output functions work, sending text to the graphics window instead of the text screen; see startdesk for details on how this works.

The endgraph function shuts down QuickDraw II.

```
startgraph(640);
/* graphics program goes here */
endgraph();
```

stderr

See stdin.

stdin stdout stderr

```
#include <stdio.h>
extern FILE *stderr;
extern FILE *stdin;
extern FILE *stdout;
```

These three predefined variables are the standard input stream (stdin), the standard output stream (stdout), and the error output stream (stderr). They are already open when any C program that runs under the shell is executing. In C programs that do not run under the shell, your program should open them using freopen before using any of these streams explicitly, or before using any of the file input and output facilities that automatically use one of these streams.

When running under the shell, the shell will automatically handle any input or output redirection. If no redirection has been specified, input will come from the keyboard, and both standard out and error out will appear on the text screen (text environment) or the shell window (desktop environment).

stdout

See stdin.

strcat strncat

```
#include <string.h>
char *strcat(char *s1, char *s2);
char *strncat(char *s1, char *s2, size_t n);
```

Strcat is a function that appends the contents of the string s2 to the contents of the string s1. A terminating null character is added to the end of the new string. Strcat returns a pointer to the first character in the string s1.

Strcat will append characters until a terminating null is found in the string s2, even if the memory allocated for s1 is not large enough to hold the completed string. It is up to the programmer to ensure that the memory allocated for s1 is large enough to hold the concatenated string and the terminating null character.

Strncat is a similar function that limits the total number of characters that can be copied. No more than n characters will be appended to the end of s1. If a terminating null character is found in s2 before n characters have been copied, all of the characters in s2, including the terminating null character, are copied to s1. If n characters are copied, and no terminating null character has been found, strncat adds a terminating null character to s1 and stops copying characters. If n is less than or equal to zero, no characters are copied.

The result is unpredictable if the two strings overlap in memory.

```
    strcat(fileName, ".obj");     /* add .obj to a file name */

    n = 64-strlen(pathName);      /* add a file name to a path name,    */
    strncat(pathName, "/", n--);  /* making sure there are no overflows */
    strncat(pathName, fileName, n);
```

strchr strpos strrchr strrpos

```
#include <string.h>
char *strchr(char *s1, char c);
int   strpos(const char *s1, int c);
char *strrchr(char *s1, char c);
int   strrpos(const char *s1, int c);
```

These functions scan a null-terminated string for a character. Strchr and strrchr return a pointer to the character if it is found. A null pointer is returned if the character is not found. The null character is considered to be a character for the purposes of the scan, so searching for the null character will return a pointer to the terminating null. The difference between the two functions is that strchr returns a pointer to the first character that matches, while strrchr returns a pointer to the last matching character.

The function strpos returns the position of the first matching character in the string. The position is the number of characters that appear before the matching character. If no matching character is found, -1 is returned.

The function strrpos returns the position of the last matching character. Like strpos, it returns -1 if there are no matches.

See also memchr.

```
    nextSpace = strchr(str, ' '); /* find the next space */

    lineLen = strpos(line, '.');  /* find # of chars in the sentence */
```

strcmp strncmp

```
#include <string.h>
int strcmp(char *s1, char *s2);
int strncmp(char *s1, char *s2, size_t n);
```

Strcmp compares two null-terminated strings. The strings are considered equal if all of the characters up to and including the terminating null character in each string matches. The string s1 is less than the string s2 if the ordinal value of the first mismatched character is smaller in s1, or if both strings match up to the terminating null character in s1, but s2 is longer than s1. Strcmp returns zero if the strings are equal, a negative integer if s1 is less than s2, and a positive integer if s1 is greater than s2.

Strncmp is similar to strcmp. The difference is that strncmp will stop comparing the strings after n characters, even if a terminating null character has not been found in either string. In that case, the strings are considered to be equal. If a terminating null character appears in either string before n characters have been processed, strncmp returns the same result strcmp would have returned. If n is zero or negative, strncmp returns zero.

See also memcmp.

```
/* scan a table to find a matching name */
for (i = 0; i < tableLength; ++i)
   if (!strcmp(name, table[i]))
      goto found;

/* scan addresses that start with a five-digit zip code, */
/* stopping if a match is found                          */
for (i = 0; i < tableLength; ++i)
   if (!strncmp(zipCode, addresses[i], 5))
      goto found;
```

strcpy strncpy

```
#include <string.h>
char *strcpy(char *s1, char *s2);
char *strncpy(char *s1, char *s2, size_t n);
```

Strcpy is a function that copies the contents of s2 to the string s1. All characters up to and including the terminating null character are copied. It is up to the programmer to ensure that the memory available for s1 is large enough to hold all of the characters, including the terminating null character.

Strncpy is similar, but it will stop copying characters after n characters have been copied. If n characters are copied and no terminating null has been found, s1 will not have a terminating null. If s2 contains fewer than n characters, s1 is padded with null characters until n characters have been moved. If n is zero or negative, no copying is performed.

Both functions return s1 as their result.

The result is unpredictable if the two strings overlap in memory.

```
   strcpy(str, "This is a test.\n");    /* copy a constant to a string */

   pos = strpos(line, ' ');      /* read the first word from a string */
   if (pos == -1)
      strcpy(str, line);
   else
      strncpy(str, line, pos);
```

strcspn strpbrk strrpbrk strspn

```
#include <string.h>
size_t  strspn(const char *s, const char *set);
size_t  strcspn(const char *s, const char *set);
char   *strpbrk(const char *s, const char *set);
char   *strrpbrk(const char *s, const char *set);
```

These functions all scan the string s and check each character to see if it is in the set of characters formed by the null-terminated string set. The set of characters can contain no characters, or all of the ASCII characters, and it can contain duplicate characters. The order of the characters in the set does not affect the scan.

The function strspn scans the string s for the first character that is not in the set. The length of the longest initial segment in s containing only characters in set is returned. If all of the characters are in the set, the effect is to return the length of the string. If the set is the null string, zero is returned.

The function strcspn performs the opposite check: it skips over characters that are not in the set, stopping when a character is found that is in the set.

The function strpbrk scans the string, skipping over characters that are in the set, and stopping at the first character that is not in the set, just like strcspn. The difference is that strpbrk returns a pointer to the character, rather than the number of characters skipped. If the string s does not have any characters from the set, NULL is returned.

The function strrpbrk works like strpbrk, except that it returns a pointer to the last character in the string that is in the set, rather than the first character. If the string s does not have any characters from the set, NULL is returned.

```
   /* find and process all of the words in the */
   /* string pointed to by line                */
   strcpy(alpha, "abcdefghijklmnopqrstuvwxyz"
                 "ABCDEFGHIJKLMNOPQRSTUVWXYZ");
   while (*line) {
      line = strpbrk(line, alpha);
      len = strspn(line, alpha);
      if (len) {
         process(strncpy(word, line, len));
         line += len;
         }
      }
```

strlen

```
#include <string.h>
size_t strlen(const char *string);
```

The function strlen returns the number of characters in a string. The number of characters is the number of characters that appear before the terminating null character.

In older versions of C, strlen returned an int, which limited the length of string that could be processed by strlen. In ANSI C, strlen returns size_t, which is unsigned long in ORCA/C. This means that strlen can effectively find the length of any string that can be held in memory. If you are porting a program that is built around the assumption that strlen is int, you can safely change the return type of strlen to int in a copy of the string.h interface file. In that case, all but the least significant two bytes are dropped from the length of the string.

```
   len = strlen(myString);
```

strerror

See errno.

strpos

See strchr.

strrchr

See strchr.

strrpos

See strchr.

strstr

```
#include <string.h>
char *strstr(char *src, char *sub);
```

The function strstr scans the string str for the first occurrence of the string sub. If a match is found, a pointer to the first character of the match is returned. If no match is found, a null pointer is returned.

```
   /* see if the correct answer occurs in the reply */
   if (strstr(reply, "Santa Fe") != NULL)
      printf("Correct!  Santa Fe is the capitol of New Mexico.\n");
```

strtod strtol strtoul

```
#include <stdlib.h>
double        strtod(char *str, char **ptr);
long          strtol(char *str, char **ptr, int base);
unsigned long strtoul(char *str, char **ptr, int base);
```

These functions convert numbers represented as ASCII strings to the internal binary format used for calculations. In each case, str is a null-terminated ASCII string. Any leading white space is skipped, and then the longest sequence of characters which can be legally interpreted as a number are read and converted. The number is returned as the function's return value.

If no conversions are possible, either because a null string was passed or because the first non-white space characters could not be interpreted as a number, zero is returned, the global variable errno is set to ERANGE, and ptr, if not NULL, is set to str. If the number causes overflow or underflow, errno is still set to ERANGE. In the case of underflow, zero is returned; in the case of overflow, infinity is returned. (Infinity is also available as the macro HUGE_VAL.)

The second parameter can be null, or it can point to a character pointer. If it is not null, and a valid string for conversion exists, it is set to point to the character immediately after the last character used in the conversion process. This is still true if an overflow or underflow occurred. If str is null, or if no valid numeric string is found, ptr is set to the value of str.

The function strtod converts a floating-point string to an extended value. The string has the same format as a string in a C source file. It can consist of a leading plus or minus sign. This is followed by a sequence of decimal digits, a decimal point, and another sequence of decimal digits. At least one of these digit sequences must appear, but either can be omitted, as can the decimal point. This is followed by an optional exponent, which, if present, consists of an e or E, an optional plus or minus sign, and a digit sequence. Imbedded white space is not allowed. Some examples of legal strings include:

```
1.0       .1        3         1e+80     1.2e-3    .1e-4     1.e17
```

The functions strtol and strtoul both have an additional parameter that is used to specify the base of the integer. The base can be zero or any number from 2 to 36. If the base is zero, the format of the number itself is used to determine the base of the number, using the same rules used by the C compiler. Namely, if the number starts with a digit from 1 to 9, it is base 10; if the number starts with 0x or 0X, it is base sixteen; and if the number starts with 0, and is not followed by an x or X, it is base 8. Any of the other values specify the base to use. For bases of 11 and higher, the alphabetic characters are used as digits, with A representing 10 and so forth to Z, which represents 35. Lowercase letters can also be used. If a letter is found which is not valid for the base in use, scanning stops. There is one exception to this rule: if the base is specified as 16, and the number starts with 0x, the first two characters are ignored. Strtol also allows an optional leading sign.

The function strtol returns a long, while strtoul returns an unsigned long. If the number is too large, strtol returns LONG_MAX or LONG_MIN, depending on the sign, and strtoul returns ULONG_MAX. In both cases, errno is set to ERANGE.

See also atof, atoi and atol.

```
   /* echo the integers in an ASCII buffer to standard out */
   while (cp != NULL)
      printf("%f\n", strtol(cp, &cp, 10));
```

strtok

```
#include <string.h>
char *strtok(const char *str, const char *set);
```

The function strtok scans the string str for tokens separated by the characters contained in the null-terminated string set. The first call to strtok should include a pointer to a string of characters for the parameter set. If the set does not change on subsequent calls, a null pointer can be passed, avoiding the overhead of reforming the set on each call to strtok.

When called, strtok starts by forming an internal representation of the set of characters. If no set was passed, the last set passed to strtok is used.

Next, if str is not null, strtok skips all characters in str that occur somewhere in the set of characters. If all of the characters are in the set, an internal pointer is set to null and a null pointer is returned. If a character is found which is not in the set, the internal pointer is set to point to the first character not in the set, and processing continues as if str had been null. By this time, though, the local copy strtok keeps of str is null, even if it was not null when strtok was called.

To continue scanning the string, subsequent calls to strtok should use a null pointer for str. The set may be changed as needed. If str is null, and the internal pointer is null, strtok returns null. If the internal value is not null, strtok skips over all of the characters that are not in the set, overwriting the first character that is in the set with a null character. The internal pointer is then updated to point to the character after the null, and the old value of the internal pointer, which now points to a null-terminated token, is returned.

```
   /* find and process words in the string pointed to by line */

   /* form the set of separator characters */
   j = 0;
   for (i = 1; i < 128; ++i)
      if (!isalpha(i))
         set[j++] = i;
   set[j] = '\0';

   /* find and process the words */
   str = strtok(line, set);
   while (str != NULL) {
      process(str);
      str = strtok(NULL,NULL);
      }
```

system

```
#include <stdlib.h>
int system(char *command);
```

The system function takes a null-terminated string as input and executes the string as a series of shell calls. The character \r can be used to separate lines if the argument consists of more than one source line. The only difference between this call and executing an EXEC file with the same commands is that an EXEC file creates a new stack frame for shell variables, while commands executed with this call work directly with the shell variables already in existence. Any command that can be typed from the shell can also be executed using this command.

As with script files, execution will halt early if any of the commands returns a non-zero error code and the {exit} shell variable is set to any value. If this happens, the error code is returned by the system function. Execution will also stop if the shell's exit command is used. The exit command also returns a value, and this value is returned by the system function. If the commands execute normally, a zero is returned.

Passing system a null pointer returns a zero if no shell is active, and a non-zero value if a shell is active.

```
   /* catalog the disk */
   system("catalog\r");
```

tan

```
#include <math.h>
extended tan(extended x);
```

The function tan returns the trigonometric tangent of the argument. If an error occurs, errno is set to ERANGE. The argument must be supplied in radians.

```
   height = tan(x)*length;
```

tanh

```
#include <math.h>
extended tanh(extended x);
```

The function tanh returns the hyperbolic tangent of the argument.

```
   n = tanh(x);
```

time

```
#include <time.h>
typedef unsigned long time_t;

time_t time(time_t *tptr);
```

The time function returns the current time as a coded integer. If the parameter tptr is not null, the time is also stored at the location indicated by tptr. If an error occurs, -1 is returned. (No error is possible in ORCA/C.) See localtime for a way to format the time in a usable fashion.

See also ctime, difftime, gmtime, mktime.

```
t = time(NULL);
```

tmpfile

See tmpnam.

tmpnam tmpfile

```
#include <stdio.h>
char *tmpnam(char *buf);
FILE *tmpfile(void);
```

The tmpnam function is used to create a file name that does not interfere with other file names on the system. If the call is successful, tmpnam returns a pointer to the file name. If buf is not null, the null-terminated file name is also stored in the character array pointed to by buf. The buffer should be at least L_tmpnam characters long. The tmpnam function can create up to TMP_MAX unique file names; under ORCA/C, TMP_MAX is 10000. Note that subsequent calls to tmpnam or tmpfile will destroy the internal copy of any file name created by tmpnam. If subsequent calls to either function will be made, and you will need the file name again, you must save the file name in your own local buffer.

The tmpfile function calls tmpnam to obtain a file name, then opens the file with flags of "w+b". The file is automatically deleted when the program stops.

```
tmpnam(myFileName);
myFile = tmpfile();
```

toascii

```
#include <ctype.h>
int toascii(int c);
```

Toascii is a macro that takes any integer argument and returns a valid ASCII character. It makes the conversion by discarding all bits except the seven least significant bits.

```
   /* make sure an integer is ASCII before checking to */
   /* see if it is uppercase                           */
   if (isupper(toascii(ch))
      printf("The character is uppercase.\n");
```

toint

```
#include <ctype.h>
int toint(char c);
```

Toint is a function that returns the value of one of the hexadecimal digits. The digits consist of the characters '0' to '9', which have the values 0 to 9, respectively; and the alphabetic characters 'A' to 'F' (or their lowercase equivalents), which have the values 10 to 15, respectively.

```
   /* convert a hexadecimal string to an unsigned long value */
   val = 0;
   len = strlen(hexString);
   for (i = 0; i < len; ++i)
      val = (val << 4) | toint(hexString[i]);
```

tolower   _tolower

```
#include <ctype.h>
int tolower(char c);
int _tolower(char c);
```

If the argument is an uppercase alphabetic character, the lowercase equivalent of that letter is returned. If the argument is not an uppercase alphabetic character, the original value is returned unchanged.

Tolower is a function. A faster macro called _tolower will perform the same operation if the argument is known to be an uppercase character, but it will also modify the argument if if is not an uppercase letter. When you know that the argument is an uppercase letter, use _tolower for efficiency. If the argument may not be an uppercase letter, use tolower.

```
   /* convert a string to lowercase letters */
   i = 0;
   while (str[i] = tolower(str[i++]));
```

toolerror

```
#include <orca.h>
int toolerror(void);
```

The toolerror function returns the error number returned by the last tool call, ProDOS call, shell call, or GS/OS call. Each call sets the error number maintained by the run-time libraries, so for an error check to be valid, it must occur immediately after the suspected call.

```
   handle = FindHandle(ptr);
   if (toolerror())
      exit(-1);
```

toupper _toupper

```
#include <ctype.h>
int toupper(char c);
int _toupper(char c);
```

If the argument is a lowercase alphabetic character, the uppercase equivalent of that letter is returned. If the argument is not a lowercase alphabetic character, the original value is returned unchanged.

Toupper is a function. A faster macro called _toupper will perform the same operation if the argument is known to be a lowercase character, but it will also modify the argument if if is not a lowercase letter. When you know that the argument is a lowercase letter, use _toupper for efficiency. If the argument may not be a lowercase letter, use toupper.

```
   /* convert a string to uppercase letters */
   i = 0;
   while (str[i] = toupper(str[i++]));
```

ungetc

```
#include <stdio.h>
int ungetc(char c, FILE *stream);
```

The function ungetc returns a character to a stream opened for input. The next call that reads a character from the file will read the character placed back into the stream by this call. Subsequent calls will read characters from the file in the normal way. The character is returned by ungetc if the call was successful; otherwise, EOF is returned. An attempt to push back EOF is ignored, and EOF is returned.

ORCA/C implements a single character buffer. An attempt to call ungetc a second time before processing the first character will result in an error.

```
   /* collect a token from a file */
   i = 0;
   for (;;) {
      ch = fgetc(myFile);
      if (isalpha(ch = fgetc(myFile)))
         name[i++] = ch;
      else {
         ungetc(ch, myFile);
         break;
         }
      }
   name[i] = '\0';
```

userid

```
#include <orca.h>
int userid(void);
```

The userid function returns the user ID assigned by the loader to a program when it is executed. The user ID returned by this call should be used whenever a toolbox call requests a user ID.

```
   printf("user id = %d\n", userid());
```

va_arg va_end va_start

```
#include <stdarg.h>
typedef char *va_list[2];

void va_start(va_list ap, ??? LastFixedParm);
??? va_arg(va_list ap, ???);
void va_end(va_list ap);
```

This collection of macros and functions allow variable argument lists to be handled from C. In a function with variable argument lists, the first call is to the macro va_start. This call is used to initialize pointers that will be used by va_arg and va_end. Two parameters are passed: the first is a local variable of type va_list, while the second is the name of the last fixed parameter in the parameter list. There must be at least one fixed parameter.

The macro va_arg is used to recover the values of the parameters. It takes the same argument variable initialized by va_start as the first parameter, and the type of the next argument as the second parameter. The value of the next parameter is returned, and the ap variable is incremented past the parameter.

The function va_end is called before leaving the function. It must be called after all of the arguments have been processed by va_arg. The va_end function cleans up the stack by removing the variable arguments from the stack.

These macros will not work if stack repair code is enabled; stack repair code is enabled by default. For an explanation of stack repair code, see Chapter 16. To turn off stack repair code, see the optimize pragma.

```
   /* return the sum of zero or more integers */

   int sum(int count, ...)

   {
   va_list list;
   int total;

   va_start(list, count);
   total = 0;
   while (count--)
      total += va_arg(list, int);
   va_end(list);
   return total;
   }
```

vfprintf vprintf vsprintf

```
#include <stdio.h>
int vfprintf(FILE *stream, char *format, va_list arg);
int vprintf(char *format, va_list arg);
int vsprintf(char *s, char *format, va_list arg);
```

These functions are basically the same as the functions with the same name, sans the leading v. The difference is that these print functions use the vararg facility (see va_arg) to specify the parameters.

write

```
#include <fcntl.h>
int write(filds, void *buf, unsigned n);
```

Up to n bytes are written from the buffer pointed to by buf to the file whose file ID is *filds*. If *n* bytes cannot be written due to lack of space, as many bytes are written as possible. The number of bytes actually written is returned. If an error occurs, a -1 is returned and errno is set to one of the values shown below.

If the file was not opened with the O_BINARY flag set, any \n characters in the buffer are converted to \r characters before being written.

Possible errors

EIO	A physical I/O error occurred.
EBADF	filds is not a valid file ID.
EBADF	The file is not open for writing.
ENOSPC	There was not enough room on the device to write the bytes.
ENOSPC	The O_BINARY flag was not set, and there was not enough room to allocate a temporary buffer for character conversion.

While this function is a common one in C libraries, it is not required by the ANSI C standard, and should be avoided if possible.

See also fwrite.

Appendix A – 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; those which are due to the restrictions imposed by the ORCA/C compiler; and those which are so serious that compilation cannot continue.

Link errors 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 a stack overflow with debugging enabled, 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

The errors listed below are all errors that the compiler can recover from. ORCA/C assigns all errors with an error level of 8. You should always eliminate every error before trying to execute a program.

`'(' expected`

The compiler expected a left parenthesis, and another token was found.

`')' expected`

The compiler expected a right parenthesis, and another token was found.

`'>' expected`

The compiler expected a greater than sign, and another token was found.

`';' expected`

The compiler expected a semicolon, and another token was found.

`'{' expected`

The compiler expected a left brace, and another token was found.

`'}' expected`

The compiler expected a right brace, and another token was found.

`']' expected`

The compiler expected a right bracket, and another token was found.

`':' expected`

The compiler expected a colon, and another token was found.

`'(', '[' or '*' expected`

The compiler expected one of these tokens, and another token was found.

`',' expected`

The compiler expected a comma, and another token was found.

`'.' expected`

The compiler expected a period, and another token was found.

`'8' and '9' cannot be used in octal constants`

Any integer constant staring with a zero is treated as an octal constant. Octal numbers do not use the digits 8 or 9.

`A character constant must contain exactly one character`

The character constant has zero characters, or more than one character. C character constants are limited to a single character. Use a string for more than one character. To create a character constant for the single quote mark, use '\''.

`A function cannot be defined here`

A function definition cannot appear in another function, a structure, or a union.

`A value cannot be zero bits wide`

A bit field was declared, and the width of the bit field was given as zero bits. Bit fields must be at least one bit wide, but no more than thirty-two bits wide.

`An #if had no closing #endif`

A preprocessor #if statement was found, but was not completed with an #endif.

`Assignment to an array is not allowed`

You cannot assign a value to an array in C, even if the value you are trying to assign is another array of the same type. Use the memcpy function to copy the contents of one array into another array.

`Assignment to const is not allowed`

An attempt has been made to assign a value to a variable that was declared as being a constant (using the const descriptor). Constant variables cannot be changed. You must either change the declaration or eliminate the assignment.

`Assignment to void is not allowed`

An attempt has been made to assign a value to void.

`Auto or register can only be used in a function body`

The auto storage class can only be used with variables defined locally in a function.

`Bit fields must be less than 32 bits wide`

ORCA/C restricts bit fields to 32 bits or less. The size of the bit field must be reduced.

`Break must appear in a while, do, for or switch statement`

A break statement was found, but it did not appear in a while loop, a do loop, a for loop, or a switch statement. The break statement is used to leave one of these structures, and cannot be used outside of the structure.

Case and default labels must appear in a switch statement

A case label or default label was found, but the label was not in the body of a switch statement. These types of labels cannot be used outside of a switch statement.

Compiler error

This error is generated when the compiler detects an internal problem. This can occur when another error appears in the program, and that error causes conflicting or missing information. This error sometimes appears before the message about the error that caused the problem. If this error appears in a program that has no other error messages, please report the problem. If it appears in conjunction with some other error, correcting the other error will get rid of this error, too.

Cannot redefine a macro

You cannot include a second definition for a macro unless the new definition is token-for-token identical to the original definition. For example, the following two definitions can legally appear in the same program:

```
#define EOF (-1)
#define EOF (-1)
```

The definitions

```
#define EOF (-1)
#define EOF -1
```

would, however, generate this error message. There are four ways to deal with the problem:

1. Remove all but one of the macro definitions.

2. Insure that each definition of the macro exactly matches all of the other definitions.

3. Undefine the old macro before redefining it.

4. Bracket the definition with conditional compilation commands to prevent redefinition, as in

   ```
   #ifndef EOF
   #define EOF (-1)
   #endif
   ```

Cannot take the address of a bit field

An attempt has been made to extract the address of a field in a structure or union that is a bit field. Bit fields are not required to start on addressable boundaries, so it is not possible to take the address of a bit field.

Cannot undefine standard macros

The standard macros __LINE__, __FILE__, __DATE__, __TIME__, __STDC__ and __ORCAC__ cannot be undefined using the #undef command.

Comp data type is not supported by the 68881

ORCA/C uses SANE to perform calculations on float, double, extended and comp numbers. When you use the #pragma float command to instruct the compiler to use the 68881 floating-point card, comp numbers can no longer be used, since the 65881 does not support the comp data type. This error message will flag any attempt to use comp numbers with the 68881 card.

Continue must appear in a while, do or for loop

A continue statement was found, but it did not appear in a while loop, a do loop, or a for loop. The continue statement is used to jump to the end of one of these structures, and cannot be used outside of the structure.

Digits expected in the exponent

A floating-point constant was coded with an e designator for the exponent, but no exponent was found. Either code an exponent or remove the e exponent designator.

Duplicate case label

Two identical values have been used for a case label in the same switch statement. Each of the case labels must be unique.

Duplicate label

Two labels with the same name have been defined in the same function. Each statement label must be different from the other statement labels in the function.

Duplicate symbol

Two symbols have been defined in the same overloading class, and the symbols were given the same name. For example, a single function cannot define the variable i as an int, and then redefine it as a float variable. One of the names must be changed so that all of the names are unique.

'dynamic' expected

The segment statement allows you to define the segment as dynamic by placing the identifier dynamic in the segment statement. This is the only token allowed in that spot. The compiler found a different token. You must remove it, or change it to dynamic.

The else has no matching if

An else statement appeared in a program, but no matching if statement was found. This can happen when the if statement is omitted entirely, when two else clauses are used with a single if statement, or when compound statements have been used improperly.

End of line expected

Extra characters appeared in a preprocessor command.

Expression expected

A token that cannot be used to start an expression was found in a position where the compiler expected to find an expression.

Expression syntax error

An expression has not been formed correctly. This can occur when an operand or operation has been omitted, or when the operands and operations are not given in the correct order.

Extern variables cannot be initialized

When a variable is declared as extern, the compiler does not create the space for the variable, it only notes that such a variable exists in some other separately compiled module or library. A variable cannot be initialized unless space is created to hold the initial value. You can, of course, initialize the variable in the single location where it is declared.

File name expected

The #include, #append and #pragma keep directive expect an operand of a file name. None was found.

File name is too long

The C compiler limits file names to 255 characters. A longer file name was found in a #include, #append or #pragma keep directive.

`Functions cannot return functions or arrays`

A function can return a pointer to a function or array, but a function cannot return another function or a whole array.

`Further errors suppressed`

When more than eight errors occur on a single line, the compiler reports the first seven, then gives this error message rather than continuing with more errors.

`Identifier expected`

The C language requires that an identifier appear at the indicated place.

`Illegal character`

A character which is not used in the C language appeared outside of a comment or string constant.

`Illegal math operation in a constant expression`

An attempt has been made to perform an operation that is not allowed in a constant expression.

`Illegal operand for the indirection operator`

An attempt has been made to use the selection operator (.) or the indirection operator (*) in a context in which it is not allowed.

`Illegal operand in a constant expression`

An operand that could not be resolved as a constant at compile time was found in an expression that must yield a constant value.

`Illegal type cast`

An attempt was made to cast a variable to an illegal type.

`Illegal use of forward declaration`

An attempt has been made to use a forward declaration in a way that is not allowed. In general, no use can be made of a forward declaration except to declare a pointer to the type.

Implementation restriction: run-time stack space exhausted

In some cases, the compiler can detect at compile time that a stack overflow will occur when a program is executed; when this happens, this error appears. This is generally caused by defining an auto array that is larger than 32K bytes. You can avoid the problem by changing the array to a static array or using a pointer to a dynamically allocated array.

Implementation restriction: string space exhausted

Each function has a buffer that is used for string constants and a few values stored when debug code is created. This buffer is limited to 8000 bytes. You must turn debugging off or split the function into two or more smaller functions to avoid overflowing the buffer.

Implementation restriction: too many local labels

The compiler generates internal labels to handle the flow of control statements. The number of labels that can appear in a single function is sufficient for about 2000 to 3000 lines of code, although the number can vary greatly based on coding style. If the limit is exceeded, you must split the function into two or more functions to avoid the limitation.

Incorrect operand size

Many 65816 operation codes require an operand of a specific size. For example, operands using indirect addressing generally require an expression value in the range 0 to 255. If the expression in the operand results in a value with an inappropriate size, and the operand cannot be legally converted to a correct size, this error will appear.

Incorrect number of macro parameters

A macro was invoked, and too few or too many parameters were supplied. Each of the parameters defined in the macro definition must be supplied exactly one time each time the macro is used.

Integer constant expected

The inline directive, used for creating tool interface files, requires an integer constant and a long integer constant for the two operands. The #pragma nda directive also requires two integer constants. Something other than an integer constant was used.

Integer constants cannot use the f designator

The f or F designator, used to force a floating-point constant to be a float rather than double value, has been used on an integer constant (such as a character or octal value). The f designator can only be used of floating-point values or decimal integer values.

`Integer overflow`

An integer constant with a value over 2147483647, or an unsigned constant with a value over 4294967295 appeared in the program. The integer is too large for ORCA/C to deal with.

`Invalid operand`

An operand was used with an instruction that does not support the operand. For example, this error would appear if you code absolute indexed addressing with a jsl instruction.

`Invalid storage type for a parameter`

Function parameters can have a storage class of auto or register. Some other storage type was specified for a parameter; it must be changed.

`Keep must appear before any functions`

The #pragma keep directive must appear before the first function definition. Move it to the top of the source file.

`L-value required`

Assignments and certain operators require an l-value. Something other than an l-value was used in one of these locations.

`Lint: missing function type`

The lint pragma has been used with bit 1 set, and the function has been declared or defined without an explicit return type.

`Lint: parameter list not prototyped`

The lint pragma has been used with bit 2 set, and a function has been declared or defined without a prototyped parameter list.

`Lint: undefined function`

The lint pragma has been used with bit 0 set, and the function has been used, but not declared or defined before the use.

`Lint: unknown pragma`

The lint pragma has been used with bit 3 set, and a pragma has been found which the compiler does not recognize.

No end was found to the string

A string constant was started, but a closing quote mark was not found.

No matching '?' found for this ':' operator

The compiler found the start of a trinary operator, but could not find the ? character that separates the two expressions.

Only one default label is allowed in a switch statement

A default label has been found in the body of a switch statement for which another default label has already been found. Only one default label can appear in any switch statement; one of them must be removed.

Only one #else may be used per #if

A #else command was found in the body of an #if statement that already has a #else clause. One of the #else statements must be removed or changed to an #elif command.

Only parameters or types may be declared here

An attempt was made to declare a variable in the declarations that follow the function header and precede the function body. Only parameters can be declared in this location.

Operand expected

The compiler expected an operand (variable, parenthesized expression, etc.) during a function evaluation, but found some other token.

Operand syntax error

An operand was used for a 65816 instruction, but the miniassembler could not compile the operand. The error message will point to the problem area.

Operation expected

The compiler expected an operation (+, *, etc.) during a function evaluation, but found some other token.

Pascal qualifier is only allowed on functions

The pascal qualifier can only be used on a function declaration or definition. This error appears when it is used on a type or variable.

Pointer initializers must resolve to an integer, address or string

The value provided in the initializer for a pointer must be an integer, an address, or a string, and the type must be correct for the pointer.

Real constants cannot be unsigned

The u or U designator, used to indicate that an integer is unsigned, has been used on a real constant. Real constants cannot be unsigned.

Statement expected

A statement was expected in the body of a function or compound statement, but a token has been found which cannot be used to start a statement.

String constant expected

String constants are required when the name of a CDA, NDA or segment is expected by the compiler. Something other than a string constant was found in one of these situations.

String too long

ORCA/C limits each individual string constant to 4000 characters. A string constant longer than 4000 characters has appeared in the source file.

Switch expressions must evaluate to integers

The operand of a switch statement must evaluate to one of the forms of integer (including characters). The function evaluates to some other type, and must be changed.

The array size could not be determined

In some instances, arrays sizes must be given, and in others, the exact size can be left unspecified. This error notes that an attempt has been made to leave the size of an array unspecified when a specific size is required.

The & operator cannot be applied to arrays

An array name is an address; applying the & operator to an address is not allowed.

The function's type must match the previous declaration

A function declaration or definition has been found for a function that has been declared earlier. This is allowed, but the type returned by the function must match the earlier declaration.

The last compound statement was not finished

The function body contains the start of a compound statement, but the source file ended before the compound statement was completed.

The last do statement was not finished

The function body contains the start of a do statement, but the function ended before the do statement was completed.

The last for statement was not finished

The function body contains the start of a for statement, but the function ended before the for statement was completed.

The last else clause was not finished

The function body contains the start of an else clause for an if statement, but the function ended before the else clause was completed.

The last if statement was not finished

The function body contains the start of an if statement, but the function ended before the if statement was completed.

The last switch statement was not finished

The function body contains the start of a switch statement, but the function ended before the switch statement was completed.

The last while statement was not finished

The function body contains the start of a while statement, but the function ended before the while statement was completed.

The number of array elements must be greater than 0

An array was declared with 0 elements. All arrays must have one or more elements.

The number of parameters does not agree with the prototype

A function call had been made, but the number of parameters supplied in the function call does not match the number of parameters declared by the function prototype. Exactly one parameter must be supplied for each parameter in the function prototype, and the types of the parameters in the call must match the types of the parameters in the function prototype.

The operation cannot be performed on operands of the type given

Not all operations can be used with all types. An attempt has been made to use an operation with an operand of an illegal type. For example, this error would occur if you attempt to use a bit manipulation operator on a floating-point number.

The selected field does not exist in the structure or union

A field has been selected from a structure or union using the -> or . selection operators, but the field does not exist.

The selection operator must be used on a structure or union

The . or -> selection operator has been used on a variable that is not a structure or union.

The structure has already been defined

An attempt has been made to redefine a structure or union. Once a set of fields has been associated with a structure or union, the structure or union tag can be used to define variables, but the fields must not be redefined.

There is no #if for this directive

A #else or #elif command has been used, but cannot be associated with a #if command.

Token merging produced an illegal token

The ## token merging operator was used in a macro, but the tokens merged did not result in a legal token. For example, merging - and 1 produces the string -1, but this string is not a legal token. (The characters "-1" form two separate tokens, the unary subtraction operator and the signed integer one.)

Too many initializers

Too many initializers have been specified for an array or structure.

Type conflict

Incompatible types have been used with one of the operators in an expression, or an attempt has been made to pass a parameter with an incorrect type to a prototyped function.

Type expected

The compiler expected a type in a declaration, but none was found.

Undeclared identifier

An identifier has been used, but it has not been declared. Except for some limited cases where a function may be called before it is used, all identifiers must be declared before they are used.

Unidentified operation code

An operation code appeared in the miniassembler, but it was not a legal 65816 operation code. Be sure the name of the operation code has not been redefined with a macro. If the error appears in an operand field, make sure the operation code requires an operand. If the error appears in a label field, be sure the label is followed by a colon.

Unions cannot have bit fields

Bit fields can only be used in structures.

Unknown cc= option on command line

There is a cc= command line option that the compiler does not recognize, possibly due to a malformed option.

Unknown preprocessor command

The preprocessor has encountered a command it does not recognize. The command must be removed.

'while' expected

A while clause was expected at the end of a do statement, but none was found.

`You cannot initialize a parameter`

Parameters cannot be initialized.

`You cannot initialize a type`

Only variables can be initialized, not an entire class of variables represented by a type.

`You must initialize the individual elements of a struct, union, or non-char array`

This error generally occurs when an initializer has been mistyped. When initializing a complex type, it is necessary to specify the values of each element or field, except for some cases when the compiler will finish initializing an area with zeros.

Terminal Compilation Errors

A terminal error encountered during compilation will abort the compile. In the text environment, if you have compiled the program from the command line 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. In the desktop environment, an attention box is displayed with the error message.

`Error purging file`

An error occurred when the compiler was releasing a file. This could be a source file or an object file created by the compiler. This error can occur due to corrupted memory, a full disk, or an error while writing the file to disk.

`Error reading file`

A file read error occurred while reading the contents of one of the source files. If an exit is made to the editor, the location may not be correct. This error is reported by GS/OS; it is generally caused by a corrupted disk or a disk with a bad block.

`ORCA/C requires version 1.1 or later of the shell`

The ORCA/C compiler has been executed from APW 1.0 or ORCA/M 1.0. You must update the shell.

Out Of Memory

There is not enough memory to compile the program.

Source files must have a file type of SRC

An attempt has been made to open a source file with a #include or #append directive, but the file was not a source file.

Terminal compiler error

An internal error has occurred that the compiler cannot recover from. This error should be accompanied by some other error. When the other errors have been corrected, this error should go away. If it does not, please contact the Byte Works.

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.

You cannot change languages from an included file

An attempt has been made to open another SRC file with a #append statement, but the file that contains the #append statement is an included file. If you are trying to do a multi-lingual compile, move the #append statement to the end of the C file that is passed to the compiler. If the file is supposed to be a C source file, change the language stamp using the Languages menu or the shell CHANGE command.

You cannot change languages with an include directive

An attempt has been made to open another SRC file with a #include statement, but the file is not a C source file. If you are trying to do a multi-lingual compile, use the #append statement. If the file is supposed to be a C source file, change the language stamp using the Languages menu or the shell CHANGE command.

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 C 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.

Appendix B – Custom Installations

This appendix is designed to help you install ORCA/C to take advantage of your specific hardware configuration. As shipped, ORCA/C 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/C environment that suits your needs. Finally, this appendix describes the principal files that make up the ORCA development environment and the C compiler; by studying this section, you can learn why we configured ORCA/C 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/C 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/C 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/C. It installs a smaller version of the ORCA/C compiler. The smaller compiler does not support precompiled headers, and the optimize pragma is ignored. Individual functions 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/C system, including the desktop development system, the text based editor, and all of the header 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/C 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/C 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/C. 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/C 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" C 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. All of the header files required by the ANSI C standard (like stdio.h) are installed, but the header files for the toolbox, ProDOS, GS/OS, and the ORCA shell are not installed. While these extra header files are not installed, you can copy any of them that you would like to use after installing the basic text system. You can use any copy program to copy the headers, which are located in the folder "Libraries/ORCACDefs" on the ORCA.C disk and the CC.Extras disk. Copy any of them that you want to the same folder wherever you installed ORCA/C. Be sure to notice that some of the header files are on the ORCA.C disk, and some are on the CC.EXTRAS disk; there isn't room for all of the headers to go on the same disk the way we ship the system.

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/C 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/C. 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/C 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/C with the ORCA/M macro assembler, installing ORCA/C 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 three 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 C system or add C to an existing ORCA/M or ORCA/Pascal system. All of the executable files from the C 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 C into an existing system that does not already have C, you will need to add the C language to your SYSCMND file. If you are installing C 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 C 1.0 system to C 2.0. It can be used to update any 2.0 level ORCA installation, but you should install ORCA/C 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 C system or add C 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 header 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 C into an existing system that does not already have C, you will need to add the C language to your SYSCMND file. If you are installing C in an existing Pascal system, be sure and follow up this step with the "ORCA Pascal, C, Asm Libraries" script, described above.

While the header files that are not required by the ANSI C standard 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/ORCACDefs" on the ORCA.C disk and the CC.Extras disk. Copy any of them that you want to the same folder wherever you installed ORCA/C. Be sure to notice that some of the header files are on the ORCA.C disk, and some are on the CC.Extras disk; there isn't room for all of the headers to go on the same disk the way we ship the system.

This installation option should not be used to update a C 1.0 system to C 2.0. It can be used to update any 2.0 level ORCA installation, but you should install ORCA/C 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 C 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. 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/C 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/C on a hard disk for effective ways of using ROM disks.

Details About Configuration

In this section, we will explore why ORCA/C is configured the way it is by looking at what happens when you run ORCA/C, 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/C 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:shell`
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 C source file, the shall takes a look at the auxtype field for the file, which will have a value of 8. The shell then scans its internal copy of the SYSCMND file looking for a language with a number of 8, and finds one with a name of CC. The shell then loads and executes the file 16:CC; 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 C compiler looks for header files. When you code a file-related C directive with brackets around the file name, like this standard include:

```
#include <stdio.h>
```

the C compiler appends this file name to the prefix 13:ORCACDefs, giving a full path name for the file of 13:ORCACDefs:stdio.h. The ORCA/Pascal compiler does something similar, but it uses 13:ORCAPascalDefs. 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 C 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 C – Run-Time License

Any program written with ORCA/C has some of C'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/C, you must include a copyright statement stating that the program contains copyrighted libraries from the ORCA/C run-time library. This copyright must appear in a prominent place. If the program has any other copyright statement, the ORCA/C copyright statement must appear in the same location. The text that must be included is:

> This program contains material from the ORCA/C
> Run-Time Libraries, copyright 1987-1992
> by Byte Works, Inc. Used with permission.

Appendix D – 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/C 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/C 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 (char) 7 to the screen beeps the speaker.

```
/* Beep the speaker */

void Beep (void)

{
putchar((char) 7);
}
```

The C language has an escape sequence that does the same thing; it is the \a escape sequence. In fact, in ORCA/C, \a maps to character 7. As a general rule, it is better to write '\a' instead of (char) 7, since the escape sequence will work if you move the program to other computers, and C programmers from other computers will recognize what your program does.

Cursor Control

The text screen normally shows the cursor as an inverse character. You can turn the cursor off by writing (char) 6, and turn it back on again using (char) 5. With some careful timing, you can even blink the cursor by alternately writing these characters. These control codes are missing in the GS/OS .CONSOLE driver.

There are several control codes that move the cursor. (char) 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.

(char) 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. The escape sequence \n maps into a character with a value of 10, and this character is used across C systems on all computers to move to the start of the next line, as opposed to moving down one line without moving to the left. To make this work, the ORCA/C libraries automatically add a \r character any time a character with a value of 10 is written to the screen. The net result is that there is no way to move down one line without also moving to the start of the line using the standard libraries; this character will move to the start of the next line, instead.

(char) 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 behaviour 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 (char) 8 is taken into account.

(char) 31 moves the cursor up one line. If the cursor starts on lin e0, nothing happens.

(char) 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. (char) 12 also moves the cursor to the top left of the screen, but is also clears the screen to all blanks. Note that (char) 12 is completely equivalent to the \f escape sequence.

(char) 13 moves the cursor to the start of the current line. It is equivalent to the \r escape sequence.

(char) 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 (char) 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,

```
printf("%c%c%c", (char) 30, (char) (10+32), (char) (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 function, below, for a convenient way of using this feature.

```
/* Move the cursor to the top left of the screen */
/* and clear the screen                           */

void formfeed (void)

{
putchar((char) 12);
}
```

```
/* Move the cursor to column x, row y */

void gotoxy (int x, int y)

{
putchar((char) 30);
putchar((char) (x+32));
putchar((char) (y+32));
}


/* Move the cursor to the top left of the screen */

void home (void)

{
putchar((char) 25);
}


/* Move the cursor to the start of the next line down */

void linefeed (void)

{
putchar((char) 10);
}


/* Move the cursor to the left */

void moveleft (void)

{
putchar((char) 8);
}


/* Move the cursor to the right */

void moveright (void)

{
putchar((char) 28);
}
```

```
/* Move the cursor up one line */

void moveup (void)

{
putchar((char) 31);
}


/* Move the cursor to the start of the current line */

void return (void)

{
putchar((char) 13);
}
```

Clearing the Screen

In the last section, we looked at (char) 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. (char) 11 clears the screen from the cursor position to the end of the screen, filling the cleared area with blanks. (char) 29 is still more selective. It clears the screen from the current character position to the end of the line.

```
/* Clear to the end of the current line */

void cleartoeol (void)

{
putchar((char) 29);
}


/* Clear to the end of the screen */

void cleartoeos (void)

{
putchar((char) 11);
}
```

Inverse Characters

Text normally shows up on the text screen as white characters on a black background. Writing (char) 15 causes any future characters to be written as black characters on a white background. (char) 14 reverses the effect, writing white characters on a black background.

```
/* Write all future characters in inverse */

void inverse (void)

{
putchar((char) 15);
}


/* Write all future characters in normal mode (white on black) */

void normal (void)

{
putchar((char) 14);
}
```

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. (char) 27 turns MouseText on, while (char) 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.

```
/* Turn mousetext on */

void mousetexton (void)

{
putchar((char) 27);
}

/* Turn mousetext off */

void mousetextoff (void)

{
putchar((char) 24);
}
```

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.

```
int main (void)

{
char ch;

for (ch = '@'; ch < '`'; ++ch) {
   putchar(ch);
   putchar(' ');
   }
putchar('\n');
putchar('\n');

mousetexton();
for (ch = '@'; ch < '`'; ++ch) {
   inverse();
   putchar(ch);
   normal();
   putchar(' ');
   }
mousetextoff();
putchar('\n');
}
```

Index