

# **ORCA/Debugger™**

A GSBug Style Source Level Debugger  
for  
ORCA Compatible High-Level Languages

**Mike Westerfield**

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



Copyright 1992  
By The Byte Works, Inc.  
All Rights Reserved

**Limited Warranty** - Subject to the below stated limitations, Byte Works, Inc. hereby warrants that the programs contained in this unit will load and run on the standard manufacturer's configuration for the computer listed for a period of ninety (90) days from date of purchase. Except for such warranty, this product is supplied on an "as is" basis without warranty as to merchantability or its fitness for any particular purpose. The limits of warranty extend only to the original purchaser.

Neither Byte Works, Inc. nor the authors of this program are liable or responsible to the purchaser and/or user for loss or damage caused, or alleged to be caused, directly or indirectly by this software and its attendant documentation, including (but not limited to) interruption of service, loss of business, or anticipatory profits.

To obtain the warranty offered, the enclosed purchaser registration card must be completed and returned to the Byte Works, Inc. within ten (10) days of purchase.

**Important Notice** - This is a fully copyrighted work and as such is protected under copyright laws of the United States of America. According to these laws, consumers of copywritten material may make copies for their personal use only. Duplication for any purpose whatsoever would constitute infringement of copyright laws and the offender would be liable to civil damages of up to \$50,000 in addition to actual damages, plus criminal penalties of up to one year imprisonment and/or a \$10,000 fine.

This product is sold for use on a *single computer* at a *single location*. Contact the publisher for information regarding licensing for use at multiple-workstation or multiple-computer installations.

ORCA/Debugger is a trademark of the Byte Works, Inc.

Byte Works is a registered trade mark of the Byte Works, Inc.

Program, Documentation and Design

Copyright 1992

The Byte Works, Inc.

Apple Computer, Inc. MAKES NO WARRANTIES, EITHER EXPRESSED OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

GS/OS is a copyrighted program of Apple Computer, Inc. licensed to Byte Works, Inc. to distribute for use only in combination with ORCA/Debugger. Apple software shall not be copied onto another diskette (except for archive purpose) or into memory unless as part of the execution of ORCA/Debugger. When ORCA/Debugger has completed execution Apple Software shall not be used by any other program.

Apple is a registered trademark of Apple Computer, Inc.

# Table of Contents

Chapter 1 – Introduction	1
The ORCA/Debugger	1
An Overview of This Manual	1
Chapter 2 – A Tutorial Introduction to the ORCA/Debugger	3
Installing the ORCA/Debugger	3
Tour of the Debugger	3
Running the Test Program	4
The Main Display	4
Help Screen	5
Step Mode	5
Switching Screens	5
Living with Subroutines	6
The Variables Display	6
Changing a Variable's Value	7
The Stack Frame Display	8
Tracing Through and Around Subroutines	8
Viewing Memory	8
Setting Break Points	9
Memory Protection	10
Entering the Debugger	10
Using Break Points	11
The Four-Finger Salute	11
Using the DebugBreak Utility	11
A Debug Script	11
Controlling Speed	12
The DebugFast Utility	12
Debugging Parts of a Program	12
Chapter 3 – ORCA/Debugger Command Reference	13
Organization	13
Entering the Debugger	13
Leaving the Debugger	14
The Command Line	14
"var:	15
"var=val	15
:	18
?	18
addr:	19
BP	19
CLR	19
FRAME	19
IN	19
INDENT	20

## ORCA/Debugger

JUMP	20
MEM	20
MP	20
OPEN file	20
OUT	21
STEP	21
TRACE	21
VERSION	21
Step/Trace Commands	21
?	22
esc	22
space	22
return	22
left arrow	23
right arrow	23
↺	23
J	23
L	23
N	23
Q	23
R	23
S	24
T	24
X	24
The Listing Display	24
Break Point Display Commands	25
?	26
esc	26
return	26
left arrow, right arrow	26
tab	27
delete	27
up arrow	27
down arrow	27
Memory Protection Display Commands	28
?	29
esc	29
return	29
left arrow, right arrow	29
tab	29
delete	30
up arrow	30
down arrow	30
Memory Display Commands	31
?	32
esc	32
return	32
down arrow	32
up arrow	33

## Table Of Contents

A	33
B	33
C	33
D	33
E	33
I	34
L	34
N	34
P	34
R	34
S	34
U	34
X	35
^ * @ &	35
Stack Frame Display Commands	35
?	36
esc	36
return	36
down arrow	36
up arrow	36
RAM Display Commands	37
?	38
esc	38
return	38
down arrow	38
up arrow	39
A	39
B	39
C	39
D	40
E	40
I	40
L	40
P	40
R	40
S	40
U	40
X	40
^ * @ &	40
Chapter 4 – Debugger Utilities Reference	43
DebugBreak	43
DebugFast and DebugNoFast	43
Appendix A – How the Debugger Works	45
COP Vector	45
COP 00	45
COP 01	45
COP 02	46

ORCA/Debugger

COP 03	46
COP 04	46
COP 05	46
COP 06	48
COP 07	48

Index	49
-------	----



# **Chapter 1**

## **Introduction**

### **The ORCA/Debugger**

The ORCA/Debugger is a powerful new debugging tool for high-level languages. It is an Init style debugger, installed as your computer boots and always sitting in the background, ready when you need it – yet in uses only 24K of memory in it's dormant state, so it loads fast and doesn't take up much of your computer's memory.

### **An Overview of This Manual**

This manual is divided into two main sections. Chapter 2 is a short, tutorial introduction to the ORCA/Debugger. It is a great way to get started, quickly reviewing the major features of the debugger, and mentioning more detailed features, pointing you to the appropriate sections of the manual if you need to use the advanced features. Chapter 2 also contains the installation instructions. I would strongly recommend that you read all of Chapter 2.

The reference manual for the ORCA/Debugger is in Chapter 3. I suggest that you don't read this chapter. Instead, skim through the pages, looking at the major topics so you know where the technical details are when you need them.

Chapter 4 is the reference manual for the utilities. Like Chapter 3, it's worth skimming so you know what information is available when you need it.

Appendix A contains detailed information about how the ORCA languages and debuggers work. These are internal details that are generally only needed by those of you who are writing compilers and debuggers of your own.



## Chapter 2

# A Tutorial Introduction to the ORCA/Debugger

### Installing the ORCA/Debugger

The ORCA/Debugger package is made up of four programs – the debugger itself and three utilities. You can install all of them using the installer.

Start by running the Installer, which is on the ORCA/Debugger disk. You will see two installation scripts, one called “ORCA/Debugger,” and the other called “Debug Utilities.” Click on the “ORCA/Debugger” line (even if it is already highlighted) and then click on install. This copies the debugger to the System.Setup folder of your System folder.

Next, click on the “Debug Utilities” script, then select your ORCA folder in the right-hand list. You will see “ORCA.Sys16” in the file list when you are in the right place. Click on install. The installer will copy the three utilities that come with ORCA/Debugger into your ORCA utilities folder, and copy three help files for the utilities into your help folder.

The last step has to be performed by hand. You need to add three lines to your SYSCMND file before the ORCA shell will recognize the three new utilities. Run the ORCA editor (or any other editor that handles text files) and load the file :ORCA.Debugger:Shell:SYSCMND from the ORCA/Debugger disk. Copy the three lines you see into the ORCA SYSCMND file. If you are using the 2.0 shell, you will find your SYSCMND file in the Shell folder, which will be located in the same folder as ORCA.Sys16. If you are using one of the older ORCA shells, the SYSCMND file will be in a folder called System. This isn’t necessarily your main system folder; the ORCA system folder is in the same folder as ORCA.Sys16.

Once you have completed installation, reboot your computer and run ORCA.Sys16. When you type help from your shell or shell window you should see the three debugger utilities listed. Their names are DebugFast, DebugNoFast and DebugBreak. If you are using Apple’s System 6 operating system, you will also see the ORCA/Debugger icon on your screen as the computer boots.

### Tour of the Debugger

The rest of this chapter is a short, tutorial introduction to the debugger, using some sample programs from the ORCA.Debugger disk. There are some debugger features we won’t talk about in this tutorial, and more that we’ll just touch on. Once you’ve finished with this chapter, glance through chapters 3 and 4 to see how they are organized so you can find the detailed technical information when you need it.

I would suggest using the text shell to run the programs as you work through these examples. The ORCA/Debugger will work from PRIZM, but we’ll be using the shell a lot, and also exploring how the text screen works when you debug a text program. Using the shell is a little easier from the text shell than it is from PRIZM. More important, some of the examples deal specifically with how the debugger interacts with the text screen.

## ORCA/Debugger

### Running the Test Program

We'll be using two programs to explore the debugger, and both are probably familiar to you. The first is the BullsEye program, which may have been the first program you ever compiled with ORCA/Pascal or ORCA/C. The other is also used to introduce you to our compilers; it's the Sort sample. Just to be indifferent, I've used the Pascal version of the BullsEye program and the C version of the Sort program, but from the debugger's standpoint, it really doesn't matter much which language you are using. Even if you only know one or the other, you should have no trouble following the examples.

### The Main Display

Let's start by jumping right into the debugger. Put the debugger disk in your computer and set the prefix to :ORCA.Debugger:Samples:, then run BullsEye. It's important to *run* the BullsEye program and not *compile* it. To do that, just type the name of the program. Assuming you've installed the debugger correctly, you'll hear a beep and see this on your screen:

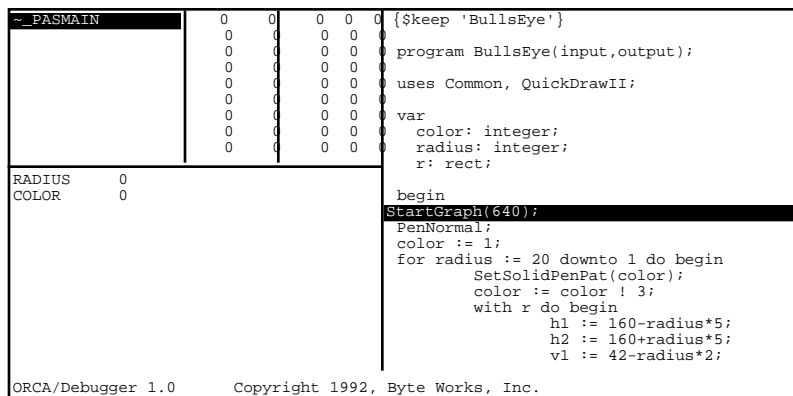


Figure 2-1: The Main Debugger Display

This is the main debugger display. It's divided up into six sections; these are called displays in this manual. Each of the displays shows you something specific about the program you are debugging.

Starting at the top left, you see the stack frame display. The stack frame display shows you a list of the subroutines that are currently running. In this case, all you see is the name of the main Pascal program, which ORCA/Pascal calls ~\_PASMMAIN. The stack frame display is used to flip back and forth among the various subroutines so you can see all of the variables in a program, even if you are debugging a recursive subroutine.

The two displays to the right are filled with zeros. The first one, with two columns of zeros, is the memory protect display. The memory protect display is used to ignore ranges of code so you can debug the part of the program you are interested in, skipping quickly over the part you don't care much about. The display with three columns of zeros is the break point display. It's used to set manual break points anywhere in your program.

Just below these three displays is the memory display. The name is stolen from GSBug in an attempt to make it easier for you to switch between the two debuggers. If you like, you can think

## Chapter 2: A Tutorial Introduction to the ORCA/Debugger

of this as the variables display. It shows all of the variables in a program or subroutine, along with their current values.

The right half of the screen shows the source code you are debugging. It's called the listing display.

The bottom two lines are the command line display. When you start, there is a copyright message in the command line display; that will go away as soon as you press a key. Don't press just any old key, though! The key you press is also used as the first key for the command you type.

### Help Screen

Start by pressing `?` and then the return key. What you get is a complete list of the command line commands. This list is a quick reminder of the commands you can use; if you don't have any idea what it's for, you can always look up the detailed information in this manual.

### Step Mode

The first command we'll try is the `STEP` command. You don't have to type the whole command name, just the `S`, followed by a return. When you do that, you enter the step/trace mode. This is where you'll probably spend most of your time in the debugger. While you are in the step/trace mode you can step through your program one line at a time, watching where you are, the values of the variables, and flipping back to the program's screen to see what the user would see. Press the space bar a few times, and you'll see how this works. Each time you press the space bar, one more line of the program executes. When you get to the bottom of the listing display, the screen scrolls.

The debugger has a lot of modes, and each has its own set of single-key commands, like the space bar command you just tried. All of the screens have two commands in common, though. The `esc` key always gets you out, returning you to the command line. The `?` key always brings up a window with a list of the commands you can use, too.

So far, you've been stepping through the program. The other half of the name for the step/trace mode comes from trace mode, which is basically when the debugger steps as fast as it can, not waiting for you to press the space bar. Press the return key to see the trace mode in action. If you get to it quickly enough, you can press the open-apple key to pause the trace – it starts again as soon as you release the key. You can also press the space bar to stop the trace, putting you back in step mode. If you weren't quick enough, rerun the program and try again.

Towards the end of the program the program's text screen kicks in. The debugger automatically switches to the program's display when text is written or read. In this case, you're near the end of the program, where the `readln` statement is found. Press return and the program will continue on.

The last of the basic stepping and tracing commands is `J`, which jumps back to your program. In fact, `J` is one of the three ways to "leave" the debugger, returning control to your program until it finishes or hits a break point. (The other two ways are to step or trace to the end of the program, or to use the `JUMP` command from the command line.) Go ahead and try the `J` command now, remembering to press return when the program executes the `readln` statement.

### Switching Screens

You've already seen that the debugger will switch to the program's screen when text is being read or written. You can also switch to the program's screen while you step or trace. The two

## ORCA/Debugger

main commands for switching the screen are `S` and `T`, which switch to the super-high resolution graphics screen and the 80 column text screen, respectively. These are the only two screens you can access under direct control from Pascal or C. You can also use the `L` key to switch to low resolution graphics, which is used by Integer BASIC.

The super-high resolution graphics screen is the 320 mode or 640 mode screen used by most Apple IIGS programs. It's the one QuickDraw and the rest of the tools use. It's also the screen used by PRIZM, so if you happen to be using PRIZM and the ORCA/Debugger at the same time, this is the screen you would normally switch to.

The 80 column text screen is the one used by the text based ORCA shell, and it's the one used in stand-alone C and Pascal text programs. It's also the one used by every other Apple IIGS shell I'm aware of, so you should have no trouble debugging text programs written for any shell.

The debugger maintains a separate text screen, distinct from the one your own program uses. The fact that both the debugger and your program can use the text screen is the reason the debugger flips back to your program screen before allowing input or output. If you pick one of the program screens, the debugger will still flip back to its own screen if you leave step/trace mode with the `esc` key, and that may be the easiest way to remember to get back to the debugger screen. You can also use the `N` key to switch from one of the program screens back to the debugger screen, though. `N` stands for "Normal" debugger display.

## Living with Subroutines

Subroutines are wonderful for organizing a program, but they do present some special problems when you are debugging a program. The most severe is that there are several different variable spaces in a program that uses subroutines. For example, you can have a global variable called `i`, and another one in each and every subroutine. Even worse, a recursive subroutine creates multiple copies of what looks like the "same" variable in the source code – one copy for each recursive call.

There's a term that goes with each of these variables spaces. Each one is called a stack frame. All of the global variables in the main program are in one stack frame, and each time a subroutine is called, it creates a separate stack frame, stuffing all of the variables that are local to the subroutine in the new stack frame.

In C, static variables are a special, half-way case. A static variable exists all of the time, like a global variable, but it is local to a particular subroutine. The debugger treats static variables as if they were normal, local variables, though. The only difference between static and local variables in C is that the static variable will always have the same value, even when the subroutine is called recursively.

## The Variables Display

The debugger's variables display shows one stack frame at a time. The variables are shown in the sort of format you would expect in a high-level language. To see how this works, run the Sort program. It's in the same folder as the BullsEye program we've been using so far.

**Note** Sort isn't exactly an uncommon name for utilities in the ORCA system – in fact, our Utility Pack #1 has a sort utility. If you have a utility named sort installed in your ORCA shell, you will need to type the full path name for the sort sample program so the shell executes the sample, and not your utility. If you aren't sure if you have a utility named sort, type `HELP` from the shell. The

## Chapter 2: A Tutorial Introduction to the ORCA/Debugger

help command lists all of the utilities. If sort isn't in the list, there won't be a conflict.

When the program starts you see two variables in the memory window. They are an integer loop counter, `I`, and an array, `A`. In the case of the integer variable you will see 0, the initial value for the variable. The array looks sort of strange, though. What you actually see is the address of the array in memory. We'll look at some commands to see all of the array a little later.

Step through the program far enough to enter the ShellSort subroutine. As soon as you hit the subroutine, the debugger switches the variables window. You still see a variable called `I`, but it's not the same one as before. You also see two other integers, `T` and `DONE`. `I`, `T` and `DONE` are the local variables defined in the ShellSort subroutine.

The `DONE` variable is actually a boolean value, but the C language doesn't have a separate data type for boolean values. Instead, C uses integers, using the convention that 0 means false and any other value means true. Still, you might want to look at `DONE` as an actual boolean value, with a True or False value. This is a simple example of a very common problem: the program displays a variable using its "natural" format, but it would be easier on you to see the value in another format. Another common example, especially in C, occurs when a character value is stored in an integer. It's also fairly common to want to see an integer value in hexadecimal format, where it is easier to read off bit flags. You can edit the memory display to change the format of any of the variables.

To edit the memory display, esc back to the command line and type `MEM`. The top variable will be inverted. Use the down arrow key to scroll down to `DONE`, then press the `A` key. This switches the display format to boolean. (`B` would have made more sense, of course, but it was already taken for single byte integers.) You can actually choose from a total of 11 different formats, and you can ask for signed or unsigned versions of any of the integers. (Try `?` to see a list of the commands.)

Scrolling with the up and down arrow keys has another use, too. It's pretty common to have more variables in a stack frame than the debugger can display. By using the up and down arrow keys, you can scroll through the stack frame to see the variable you are interested in.

### Changing a Variable's Value

When you start to debug a new subroutine, it's pretty common to find an error that sets a variable to a wrong value – or doesn't set it at all, when the variable should have been initialized. You could back out of the debugger, change the program, recompile it, and start testing again. You could also change the value in the debugger and keep stepping through the program looking for more errors, though, and that just might save you a lot of time.

Use esc to get back to the command line display. Now type

```
"I=10
```

and press return. The `"` tells the debugger that you are typing a variable name next, and the `I` is, of course, the name of the variable. Be sure you type an uppercase `I`, though. Variable names are case sensitive, so `"i=10` won't do. The command sets the value of `I` to 10.

The debugger is well aware of the format for each of the variables, and expects you to use the same format when you type a value. For example, `DONE` started out as an integer value, but we changed its display format to boolean. As long as the display format is boolean the debugger expects you to type any new value as boolean, too. For example,

## ORCA/Debugger

```
"DONE=true
```

will work just fine, but if you try

```
"DONE=1
```

the debugger will beep to flag the error, and ignore the new value. If you want to set the value of `DONE` using an integer, you have to change the display format to integer first.

The rules for typing variable values should be pretty familiar to you, since, for the most part, they match the rules used in Pascal and C. If you want detailed information, though, it's all broken down very carefully in Chapter 3.

## The Stack Frame Display

There are two global variables that we can't see at the moment. The stack frame display gives you a way to see them.

From the command line, type `FRAME`. This puts you in the stack frame display editing mode, where you have very few options. Basically, you use this mode to scroll through the various stack frames. If you press the up arrow key, you will see the global variables stack frame. Press the down arrow key to move back to the local variable display.

## Tracing Through and Around Subroutines

Right at the moment, you're in the middle of the `ShellSort` subroutine. Let's assume you're finished seeing what you wanted to see in this subroutine, but you want to continue tracing from the main program. Type `R`, and you will pop out of the subroutine. Actually, the debugger ran the rest of the subroutine, stopping when it got back to the main program. Technically, it ran through the subroutine, returning to the caller – which, in a large program, probably would have been another subroutine.

Step through the program carefully until the cursor is on the line that calls `BubbleSort`. Let's assume you know this subroutine works, and don't want to bother stepping or tracing through the subroutine. Press `x`, and the debugger steps to the next line, running `BubbleSort` at full speed. This also works with function calls. Now press `x` a few more times. As you can see, except for the fact that it steps through subroutines, the `x` command works just like the `STEP` command (space bar).

## Viewing Memory

Run `Sort` again. The value of `I` is pretty clear, but there's still the array. From the command line, type

```
"A:
```

What you see is a full memory dump, like the one in Figure 2-2.



## Chapter 2: A Tutorial Introduction to the ORCA/Debugger

A										
+0	20	19	18	17	16	15	14	13	12	11
+10	4265	-28696	-5003	-22259	3560	30351	3564	-13438	-20734	-6043
+20	-14067	-6039	13264	26287	3560	-5943	-12275	-20694	-6011	-1406
+30	-2	8656	-1528	26746	-9638	18472	-14008	10	2768	3572
+40	-24064	6156	34	-7936	3234	8728	0	26849	-20629	-6041
+50	18445	26031	3560	-23736	18440	34	0	-1397	26746	-9638
+60	27563	-6039	13	-4850	13	0	0	0	0	0
+70	0	0	0	0	0	-1	530	-2	0	0
+80	0	0	0	0	0	0	0	0	-1	530
+90	-2	8264	-5574	23656	-3204	-29939	-21685	31482	244	-3072
+100	0	244	23040	-21030	-3302	-21176	-3304	15176	23307	-2826
+110	-25366	-5485	-27358	3562	421	773	976	22402	6145	421
+120	2153	-31488	-28671	-6654	-24061	0	-7525	-18656	-4095	-1404
+130	-4064	-14072	-4062	-14076	-12279	-14333	-4736	-18456	-4095	-1406
+140	-4064	-13852	-4062	-13856	-4087	-14116	-4736	8386	6794	2570
+150	14477	23274	25400	31233	18504	244	18432	-10833	3562	-3000
+160	-16376	244	-3072	0	674	8713	0	-28447	-32732	3342
+170	30031	8308	26223	27936	28005	29295	-2951	13	16116	-2383
+180	6924	34	-7936	-87	23807	-3204	26637	2949	-31384	-2456
+190	2	2999	-22614	-29429	-5580	13966	6378	14445	-28438	-6143
+200	2949	3462	8418	160	-18688	-26879	-4085	-14333	-2176	8386
+210	3493	901	2981	389	13997	-31254	-21235	-5580	2949	423

Figure 2-2: RAM Display

This is the RAM display, used to look at large chunks of memory using any one of the variable formats. In this case, it's a handy way to look at the contents of the array A. You can switch the format used to display the memory with the same keys you used to change formats in the memory display. (Press ? for the help screen for a quick summary.)

### Setting Break Points

A break point causes the debugger to stop. Break points are set on a particular line of the source code, and when the program gets to that line, the debugger breaks. If you are already in the debugger, doing a trace or running through a subroutine, the debugger simply beeps and stops the trace, leaving you in step mode. If you are running the program at full speed when the break point is hit, the debugger kicks in, showing the debugger screen and letting you enter debugger commands.

There are two kinds of break points. Hard-coded break points are the ones you set from an editor. These break points are placed in the executable program by the compiler. The debugger will always stop when it hits a hard-coded break point.

The other kind of break point is a manual break point you set from within the debugger. Manual break points are a lot more flexible than hard-coded break points, but you have to spend a little more time setting them up, too.

To set a manual break point, start by typing BP from the command line. This edits the break point display. The break point display is the one with three columns of numbers, just to the left of the listing display. When you edit the break point display, the first number on the first line is highlighted, and the highlighting of the listing window goes away. You're ready to pick out the line where the break will occur. Use the up and down arrow keys to move through the source file until the line you want to break at is highlighted. You can move a little faster by holding down the open-apple key while you press the arrow (this moves 10 lines at a time), by holding down the option key while you press the arrow (moving 100 lines) or by holding down both (which moves 1000 lines).

Most large programs are split across several source files. Use the tab key to switch between the various source files to find the one you want. If the debugger hasn't already loaded the source file you want, you'll have to use the OPEN command (described in Chapter 3) to load the source file before you start editing the break point display.

## ORCA/Debugger

The second column is a trigger value. Manual break points don't always break right away. Instead, you can set a trigger count, and the debugger won't break until the line with the break point has been executed that number of times. This is real handy when you want to break inside a loop, but you want to see the last few executions, and not trace through all of the executions of the loop. To set the trigger value, use the right or left arrow key. That moves the cursor to the trigger value. Now use the up and down arrow keys to select the proper trigger value. If you like, you can hold down the open-apple key to step the value a little faster.

The last column is the number of times the break point line has actually been executed. The debugger sets this value to 0 when you change a break point, and increments it by one each time the break point line is executed. As soon as the execution count is equal to the trigger value, the debugger breaks.

There are several commands that can help you manage manual break points. `IN` and `OUT` let you tell the debugger to ignore the manual break points, and `CLR` lets you clear the manual break points. For details about these commands, see Chapter 3.

## Memory Protection

The memory protection display is just to the left of the break point display. It's the one with two columns of numbers. Memory protection lets you tell the debugger to execute a series of lines at full speed. There are several times when this is handy. One is to ignore entire subroutines that you already know work. Another is to block of a time-consuming chunk of code in a subroutine, such as a loop that is initializing an array. Either way, the debugger will execute the lines you've blocked off at full speed.

Setting a memory protection range is a lot like setting a break point. Type `MP` to edit the memory protection display. Use the arrow keys in conjunction with the option and open-apple keys to select the first line of the area you want to protect, then use the left or right arrow key to move to the second column, and select the end of the area you want to protect.

## Entering the Debugger

The ORCA/Debugger works together with the compiler to create a program and a debugging environment that work with one another. While there are several ways to enter the debugger, all depend on the compiler creating the debug code necessary for the debugger to realize there is a program to debug. There are a variety of ways to create the debug code in the first place.

From the PRIZM desktop development environment, you can select a compile option that tells the compiler to create debug code. You do this by selecting the debug code check box in the Compile dialog. If you compile and execute the program from PRIZM, though, PRIZM's debugger will be used to debug the program, not the ORCA/Debugger. If you are using PRIZM to compile the program, but you want to use the ORCA/Debugger to debug it, be sure you turn off the execute after linking option in the Link dialog, then execute the program either as an S16 program, or by leaving PRIZM and running the program from the text shell.

From the text development environment (or from a script) you can create a program with debug code by using the `+D` flag on your compile line. The `+D` flag is supported by all of the ORCA shell commands that compile a program, including those that also link and execute the program. For example,

```
cmpl +d program.cc keep=program
```

will create a program that you can debug with the ORCA/Debugger.

## Chapter 2: A Tutorial Introduction to the ORCA/Debugger

ORCA/C also has a debug pragma. You can use this pragma to create debug code without using flags or setting options in PRIZM. See the ORCA/C manual for details.

### Using Break Points

The method we've used to get into the debugger so far is a hard-coded break point. Hard-coded break points are set in an editor; as this manual is written, the only editor that supports them is the PRIZM desktop development environment. Even if you're a die-hard text interface fan, though, you can install PRIZM as a utility and still use it just to set break points and auto-go points. (Auto-go points are like memory protection ranges, but they are hard-coded.)

Once you set a manual break point, just compile the program in the normal way, with debug code turned on. You will pop into the debugger as soon as one of the lines with a break point is executed.

### The Four-Finger Salute

For long programs, you can get into the debugger by holding down the option, open-apple and shift keys, then pressing the tab key. That's called the four-finger salute.

This method works pretty well for long programs, but there is a problem: the debugger has to actually execute a line of code while this keypress is in the keyboard buffer before it will kick in. There are a lot of situations where your program is either busy doing something besides executing the lines you wrote, or where your program is also reading keypresses and is getting them before the debugger. In those situations, you may find it difficult or even impossible to get into the debugger with the four-finger salute. Some common examples of this situation are when your desktop program is in the event loop; when your text program is waiting for keyboard input; or when any program is doing something time consuming, like reading or writing a disk file.

### Using the `DebugBreak` Utility

The last way to get into the debugger is the `DebugBreak` utility. You can run this utility anytime you want – say, in the script that builds your program – and the debugger will break into your program on the first executable line. The utility sends a message to the debugger, telling it to treat the next executable line as if it were a hard-coded break point. Since no other programs besides the one you are developing will be using debug code, nothing bad happens even if your program doesn't get executed. In fact, you can even use the `DebugBreak` utility while you are in the shell, then back out to another program launcher before you run your program.

### A Debug Script

Of course, you might prefer just running a debugger as a utility. There isn't a utility version of the ORCA/Debugger, but the `DebugBreak` utility gives you a way to create one with just a couple of lines in a script file. Here's the script file for entering the debugger as if it were a utility:

```
DebugBreak
{Parameters}
```

That's all there is to it. Create this as a script file (language type EXEC) and save it in your utility folder with a file name of, say, `DEBUG`. Next, add the line

## ORCA/Debugger

DEBUG            U                    Utility style debugger

to your SYSCMND file. That's the same file you added the debugger utilities to back when you installed the ORCA/Debugger. If you like, you can even add a help file.

To use the script, just type

DEBUG file

where file is the name of the program you want to debug. The script will even support command line parameters, so you can use it to debug shell utilities.

## Controlling Speed

One of the problems with an intrusive debug mechanism like the one used by the ORCA languages is that it slows down the execution of the program, even when you are not using the debugger. For programs that spend most of their time doing math or calling the tools, the time difference isn't that big, and may not be a factor while you are developing the program. For programs where it is a problem, though, there are two alternatives to repeatedly recompiling the program with and without debug code.

### The DebugFast Utility

The DebugFast utility tells the debugger to replace debug code with JMP instructions as the program runs. This will speed up all but the tightest code to the point where the program runs almost as fast with debug code as without. For details, refer to Chapter 3.

## Debugging Parts of a Program

You don't have to put debug code in the entire program. If your program is broken into more than one source file, it's perfectly OK to compile some of the source files with debug code turned on, and some with debug code turned off. If you are using C, you can even use the debug pragma to turn debug code on and off for individual subroutines. Just be sure that each subroutine is consistent – in other words, don't turn debug code on or off inside of a subroutine.

The only disadvantage to turning debug code off for some source files or subroutines is that you will lose access to the global variables and to the stack frames of any subroutines that called the ones with debug code on. You can get access to the global variables back, though, by making sure the program body of your Pascal program, or the main() function in your C program, is one of the subroutines compiled with debug code on.

# Chapter 3

## ORCA/Debugger Command Reference

### Organization

The ORCA/Debugger is a permanent initialization file. It is installed automatically by the GS/OS operating system when your computer boots, and stays in memory at all times. You enter the debugger in one of several ways while the program you want to debug is in memory. Once you are in the debugger, you will see the debugger's main text display. At this point, you can enter any of the commands listed below in the section "The Command Line." Several of these command line commands let you edit the various displays, or even switch to completely different screens. Each of these displays has a different set of operations; the commands you can use are described in the following sections based on the display you are editing.

### Entering the Debugger

There are three ways to enter the debugger:

1. You can set a hard-coded break point in the source file, then compile the source file with debug code turned on. The program will break into the debugger when it hits the first break point.

Some editors can be used to set a break point, including the PRIZM desktop development environment.

2. You can compile a program with debug code turned on, then run the `DebugBreak` utility. This utility tells the debugger to break into the program as soon as you start to run the program. See Chapter 4 for details on using the `DebugBreak` utility.
3. While any program that has been compiled with debug code is running, hold down the option, open-apple, and shift keys, then press the tab key. The debugger will break into your program at the currently executing line. This is called the four-finger salute.

If your program is polling the keyboard or fetching events from the Event Manager when you press these keys, your program may intercept the keypress before the debugger sees it. Try pressing the key a few more times.

If your program is doing some time-consuming task using GS/OS, the tools, an assembly language library, or a part of your program that does not have debug code turned on, you won't break into the debugger until the task is complete.

## ORCA/Debugger

If your program has called a library routine to read a line – such as Pascal’s `readln` or C’s `scanf` – you won’t be able to enter the debugger until your program has read the line. In some programs, you may be reading lines from the keyboard so fast that it will be impractical to get into the debugger using these keys.

## Leaving the Debugger

The debugger starts by interrupting your program, and you leave the debugger by telling it to start running your program at full speed. From the command line, use the `Jump` command. From the Step/Trace mode, press `J`.

## The Command Line

Figure 3-1: The Command Line Display

The command line is a line at the bottom of the screen where you can type commands. It’s the command line display that is active when you first enter the debugger. A copyright message fills the command line, but it will vanish as soon as you press a key. The key you press is also the first letter for the first command you type.

There are quite a few commands that you can use from the command line, but they fall into three broad groups. The first group of commands execute the program in a variety of ways. These commands either return control to your program or put you in the step/trace mode, described in the next section. The next group of commands let you edit the various other displays to set break points, scroll the variables display, and so forth. Each of the displays has a separate set of keystrokes that effect that display; these are described by display in the sections later in this chapter. The last set of commands are those that do something immediately in the command mode.

Table 3-1 shows the various command line commands. The characters you must type exactly are shown in uppercase, although you can use either uppercase letters or lowercase letters when you actually type in a command. Lowercase letters are used for fields that change from one time the command is typed to the next. For example, the `INDENT` command is used to scroll the listing window horizontally. You have to type the name of the command as is, but the number of columns you want to indent can change, so it’s shown as a lowercase `n`.

## Chapter 3: ORCA/Debugger Command Reference

You don't have to type the entire command name to execute a command, just enough letters to distinguish the command from the others. For example, `S` will execute the `STEP` command. In the case of a tie, the command that is first in alphabetical order is executed. This means the `IN` command is executed if you type `I` or `IN`, and the `MEM` command is executed if you just type `M`.

<code>"var:</code>	Display RAM starting at the location of a variable.
<code>"var=val</code>	Set a variable to a new value.
<code>:</code>	Display RAM from the last address used.
<code>?</code>	Show the command line help screen.
<code>addr:</code>	Display RAM starting at an address.
<code>BP</code>	Edit the break point display.
<code>CLR</code>	Clear all manual break points.
<code>FRAME</code>	Edit the stack frame display.
<code>IN</code>	Use manual break points.
<code>INDENT n</code>	Indent the source listing window.
<code>JUMP</code>	Jump back to the program in real-time mode.
<code>MEM</code>	Edit the memory (variables) display.
<code>MP</code>	Edit the memory protect (auto-go) display.
<code>OPEN file</code>	Open a source file.
<code>OUT</code>	Don't use manual break points.
<code>STEP</code>	Enter step/trace mode in step mode.
<code>TRACE</code>	Enter step/trace mode in trace mode.
<code>VERSION</code>	Show the version number and copyright.

Table 3-1: The Command Line Commands

### **"var:**

A separate RAM display is shown, with the name of the variable you enter shown at the top of the screen. The screen is filled with as many variable values as will fit, starting with the variable you specified. You can change the format used to display the variables or scroll through memory to see other values. For details on the commands you can use, and a complete description of the various variable formats, see "RAM Display Commands," later in this chapter.

When the screen is first displayed, the debugger will use the same number format that is currently used to display the value of the variable in the variables display.

The variable name itself must be typed exactly as it appears in the variables display. The variable name is case sensitive, so if the variable name in the variables window is uppercase, you must type the name of the variable using only uppercase letters.

### **"var=val**

This command lets you change the value of a variable while your program is running.

The variable name itself must be typed exactly as it appears in the variables display. The variable name is case sensitive, so if the variable name in the variables window is uppercase, you must type the name of the variable using only uppercase letters.

The value of the variable can be typed in a variety of formats. The debugger expects you to use the same format to enter a variable as the one used to display it in the variables window. The debugger trusts you – and if you abuse that trust, you can easily cause the computer to crash. For example, if you change the format of a two-byte integer to an extended real value, then set the

## ORCA/Debugger

value using this command, you will change ten bytes of memory. If the extra eight bytes happen to be used for other variables, you will see those variables change, too. If the extra bytes are used for executable parts of your program, or as part of some other program, you could cause the computer to crash. Since the memory you just wiped out may not be executed for a while, the crash may not occur right away.

You can only change the value of a variable in the current stack frame. If there are a lot of variables in a stack frame, not all are always visible in the variables display. That's not a problem – you can set the value of a variable whether or not the debugger has room to show the value in the variables display. If you want to set a variable from a nested subroutine, or set a global value, though, you have to switch to the proper stack frame before you can set the variable's value.

### Integers

Integer values are entered as signed decimal values, or as hexadecimal values.

Signed integer values can range from -2147483648 to 4294967295. If you are familiar with the way integers are stored, you can see that these values allow you to enter any legitimate signed or unsigned value for a long integer. Some of the values overlap – the unsigned value 4294967292 is stored the same way as the signed value -4. When you type in the value, the debugger doesn't care which one you use. Either one will set the bits in memory the same way. The display format you have selected will determine if the variable is displayed as a signed or unsigned value in the variables window.

A leading + sign is allowed, but not required.

Hexadecimal values are entered as a \$ character followed by zero to eight hexadecimal digits. If fewer than eight digits are entered, the value is padded on the left with zeros. For example, \$FF is a valid value. It is treated exactly like \$000000FF.

The debugger does not check for overflows. If you type a number that is outside the range of valid four-byte integers, the debugger will use the least significant 32 bits of the value and continue on without flagging an error. All integer values are treated as four-byte values. The debugger then extracts the least significant word if you are setting the value of an integer, or the least significant byte if you are setting a byte. For example, \$0101 is equivalent to 257, which will fit in either an integer or long integer variable. If you set a byte to \$0101, though, only the least significant byte is used. The least significant byte is \$01, so the byte variable would be set to 1.

Examples:

1            400534 +38        -16        \$00a6    \$FFFFFFFFE

### Real Numbers

Real numbers are entered in the following format:

<sign><digits>.<digits>e<sign><digits>

The leading sign is either a '+' or '-', and can be omitted for positive numbers.

The number itself is a sequence of digits with a decimal point. If there is no fractional part, you can omit the decimal point. The decimal point can also appear before or after all of the digits in the number.



## Chapter 3: ORCA/Debugger Command Reference

The exponent starts with the character 'e' or 'E'. You can omit the exponent, but if you use one, you must have at least one digit after the 'e'. The exponent sign is either a '+' or '-', and can be omitted for positive numbers.

The valid range of numbers depends on whether you are setting a single-precision real number, a double-precision real number, or an extended number. The approximate range for each format is:

<u>precision</u>	<u>approximate range</u>
single	1e-38 to 1e38
double	1e-308 to 1e308
extended	1e-4932 to 1e4932

If you enter an illegal value you will hear a beep, and the variable will not be set. If you enter a value that is outside of the range of values that the variable can hold, the value will be set to 0 or infinity (displayed as INF) as appropriate.

Examples:

```
3.14    0        .4      1e-673  -5.3
```

### Characters

Characters are entered as a quote mark followed by a character. If you omit the character, the debugger assigns chr(0). In all other cases, the debugger assigns the first character that follows the quote mark.

You can use either a single quote mark or double quote mark at the start of the character. No closing quote is required.

There are many valid character values that cannot be typed from the keyboard. These can be entered as integer values by first switching the variable to a one-byte integer, setting the value, then switching back to a character format.

Examples:

```
'2      "G      ' "
```

### Pascal Strings and C Strings

Strings are entered as a quote mark followed by zero or more characters. All of the characters from the first one after the quote mark to the end of the line are placed in the string. You should not use a closing quote mark – if you do, the debugger will use it as a character in the string.

You can use either a single quote mark or double quote mark at the start of the string.

The debugger has no way of checking to see if a string can fit in the space set aside for the value by the program. If you create a program with a 10 character string space, then set the string to a 40 character value in the debugger, you will wipe out 30 bytes of memory after the string. In short, it's up to you to make sure that the value you type will fit in the available space.

C strings and Pascal strings are entered the same way. Pascal strings are stored in memory with a leading length byte. C strings (and Standard Pascal strings) are stored in memory with a trailing null character.

Examples:

```
'Hello, world.  
"McCoy said tonelessly, "He's dead, Jim."
```

## ORCA/Debugger

### Boolean Values

Pascal uses a separate type for boolean values. The debugger allows you to set these boolean values by typing “true” or “false,” just as you would in a Pascal program. The debugger isn’t very picky about what you type, though. Anything that starts with the character ‘f’ is accepted as false, while anything starting with ‘t’ will do for true.

Setting a boolean value to true sets a single byte of memory to 1, while setting a value to false sets a single byte to 0. In some cases, the boolean variable may be two bytes long, and you may need to set both bytes. In that case, switch to an integer format for display and set the integer to 0 or 1, then switch back to a boolean display. C programmers may also want to display a value as boolean while debugging, but test the program with values for true other than 1. Again, to set some other value, convert the format to integer.

Examples:

```
true  False  TRUE  f
```

### Pointers

There are two steps to changing the location a pointer variable points to. The first step is to display the pointer variable as a pointer. If you try to set a value while the variables window shows the value being pointed to, you will change the value, not the pointer itself. The second step is to actually change the pointer’s value.

For example, let’s say a pointer variable is set to \$04/0046, that the two bytes at \$04/0046 happen to hold 367, and that the pointer points to an integer. If you see 367 on the screen, you are looking at the value being pointed at. You can change the value of 367 at memory location \$04/0046, but you can’t change the value of the pointer itself. If you see \$04/0046 in the variables display, you can change the pointer itself.

pointers are entered as hexadecimal values with an optional / character separating the bank byte from the rest of the address. You don’t have to enter leading zeros, and any extra digits are stripped. In fact, all of the addresses shown in the examples below are for the same value of \$04/0046.

Examples:

```
04/0046      0004/0046      040046      4/46
```

:

Using the : character as a command shows the last RAM display you viewed, with the same format, offset, and starting location. You would normally use this command to quickly flip back to a RAM display that you had set up with the `addr:` command or the `"var:` command. See the description of either of those commands for more details.

?

The help command brings up a dialog with a brief listing of the command line commands. The ? key is used throughout ORCA/Debugger; you can always type the ? character to get a list of the commands you can use.

## Chapter 3: ORCA/Debugger Command Reference

### **addr:**

A separate RAM display is shown, with the address you enter shown at the top of the screen. The screen is filled with as many variable values as will fit, starting at the address you specified. You can change the format used to display the variables or scroll through memory to see other values. For details on the commands you can use, and a complete description of the various variable formats, see “RAM Display Commands,” later in this chapter.

The address is given as a hexadecimal value, with or without a / character between the bank number and the least significant word of the address. For example,

E12000:

would display the contents of the super high resolution graphics screen. You could also use

E1/2000:

to display the same thing. You can omit leading zeros in either field, so

1/400:

will display the bank 1 portion of the text page. The full address for the text page is 01/0400.

### **BP**

The BP command lets you edit the break point display, which is used to set manual break points. For details, see “Break Point Display Commands,” later in this chapter.

### **CLR**

The CLR command clears all of the manual break points you’ve set in the BP display. This actually removes the values, setting all fields in the display back to 0. This will speed up tracing and execution slightly, since the debugger doesn’t have to check each line to see if it has been flagged as a manual break point.

See also IN, OUT.

### **FRAME**

The FRAME command lets you edit the stack frame display, which is used to switch between the various stack frames to look at local variables during nested subroutine calls. For details, see “Stack Frame Display Commands,” later in this chapter.

### **IN**

The In command reverses the effect of the OUT command. After executing the IN command, manual break points are no longer ignored.

See also CLR, OUT.

## ORCA/Debugger

### **INDENT**

The `INDENT` command lets you shift the source code shown in the listing window to the left or right.

The debugger doesn't follow indenting in the source code automatically, and can only show 40 columns of source code at a time. If you are tracing through a section of code that has been indented a lot, you can use this command to shift the display over a few columns so you can see the source code a little better.

The `INDENT` command takes a single integer as a parameter. When the source window is drawn, the debugger skips that number of columns before it starts to display letters. For example,

```
INDENT 16
```

will cause the display to shift over 16 characters.

```
INDENT 0
```

is the default, which doesn't skip over at all.

### **JUMP**

The `JUMP` command shuts down the debugger and starts executing your program at full speed. Manual break points remain in effect, so triggering a manual break point will restart the debugger. You can also restart the debugger with the four-finger salute or by hitting a hard-coded break point.

### **MEM**

The `MEM` command lets you edit the memory display, also called the variables display. The memory display shows the names and values of several variables. By editing the memory display, you can scroll the display to show other variables or change the format used to display the variables. For details, see "Memory Display Commands," later in this chapter.

### **MP**

The `MP` command lets you edit the memory protect display, which is used to tell the debugger to execute specific lines of the program faster. This capability is called auto-go in the PRIZM desktop debugger. For details, see "Memory Protection Display Commands," later in this chapter.

### **OPEN file**

The `OPEN` command loads a source file from disk and places it in the list of files.

The debugger always shows the active source file (the one where execution is taking place) so the source file you load is immediately hidden. The reason for this command is to help you set manual break points and memory protection ranges. In both cases, you need to see the source file to select the correct line. Once you have loaded a file with the `OPEN` command (or once the file is loaded automatically by the debugger) you can use the tab key to move to the proper file.

For more details on setting break points in multi-file programs, see "Break Point Display Commands," later in this chapter. For details on setting memory protection ranges, see "Memory Protection Display Commands," later in this chapter.

## Chapter 3: ORCA/Debugger Command Reference

### **OUT**

The **OUT** command tells the debugger to ignore all of the manual break points you have set in the **BP** display. This speeds up traces and program execution slightly, since the debugger no longer has to check each line to see if it is a manual break point.

The break points are not removed, just ignored. You can enable the break points using the **IN** command. Break points will also be enabled if you edit the **BP** display and make any changes. (If you just enter the **BP** editing mode and leave right away, break points will not be enabled. You actually have to make some change to enable the break points.)

See also **CLR**, **IN**.

### **STEP**

The **STEP** command enters step/trace mode in step mode. The next command to be executed is highlighted, and you can use any of the single-key step/trace mode commands. For a complete list of the commands, see “Step/Trace Commands,” later in this chapter.

### **TRACE**

The **TRACE** command enters step/trace mode in trace mode. Your program is executed as fast as the debugger can execute it. You can use any of the single-key step/trace mode commands listed in “Step/Trace Commands,” later in this chapter.

### **VERSION**

The name of the program, version number, and copyright are shown on the command line. As soon as you press a key, the command line is cleared. The key you press becomes the first key typed for the next command.

The debugger executes this command automatically when you enter the debugger.

## **Step/Trace Commands**

The step/trace mode is the main debugger mode. You can use the step/trace mode to single step through a program, trace a program and watch the various variables update, step through subroutines, or step to the end of a subroutine. You can also flip between the various displays while you step or trace.

Table 3-2 shows the single-key commands that are available while you are in step/trace mode. Character commands are shown in uppercase, but the command processor is not case sensitive. You can use the equivalent lowercase letter if you prefer.

## ORCA/Debugger

<code>?</code>	Display a help window with this list of commands.
<code>esc</code>	Leave step/trace mode and return to the command line mode.
<code>space</code>	If you are not already in step mode, start stepping. If you are in step mode, step one line.
<code>return</code>	Start tracing the program.
<code>left arrow</code>	Slow down the trace.
<code>right arrow</code>	Speed up the trace.
<code>⌘</code>	Pause a trace. The trace resumes when the key is released.
<code>J</code>	Switch back to real time execution.
<code>L</code>	Display the low resolution graphics screen.
<code>N</code>	Display the debugger's screen.
<code>Q</code>	Turn sound on or off.
<code>R</code>	Go until the next return.
<code>S</code>	Display the super high resolution graphics screen.
<code>T</code>	Display the 80 column text screen.
<code>X</code>	Run through subroutines.

Table 3-2: The Step and Trace Commands

### **?**

The `?` key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### **esc**

The `esc` key leaves the step/trace mode, returning you to the command line mode.

### **space**

If you are tracing, running through a subroutine, or running until the next return, pressing the space key puts you in step mode. Your program is frozen while you are in step mode, waiting for you to execute some command.

If you are already in step mode when you press the space bar, the debugger executes one line in the program, then stops, waiting for another command.

One way to trace slowly through a program is to press and hold down the space bar. The repeat feature of the Apple IIGS will kick in, effectively pressing the space bar very fast.

### **return**

Pressing `return` starts tracing the program. The debugger still updates the debug display while tracing, which takes time, but trace mode is the fastest way to execute the program and still be able to watch the cursor scroll through the source listing and the variable values update in the variables display. The debugger runs the program as fast as it can while still updating the displays.

There are several things that effect the speed of tracing. Subroutine calls slow the debugger down in general, and redrawing the stack frame display also takes a little time, so portions of a program that are tracing through subroutine calls will run slower than portions that don't. Scrolling the source window also takes time, so tight loops that fit entirely on the screen trace

## Chapter 3: ORCA/Debugger Command Reference

faster than code that scrolls the display. Finally, updating the variables window takes more time if there are more variables. It also takes a lot of time to format and print the value of a floating point variable.

You can speed up or slow down the trace using the left and right arrow keys, and pause a trace by holding down the open-apple key.

### **left arrow**

Slows down a trace by adding a small delay each time a line is executed. By default, the debugger traces as fast as possible, with no delay.

### **right arrow**

Speeds up a trace by removing the small delays inserted by the left arrow key.



Pauses a trace. The trace resumes as soon as you release the open-apple key, but you can type one of the other commands to stop it right away. For example, pressing the space bar while you hold down the open-apple key will put you in step mode.

### **J**

Using this command is one way to leave the debugger. The debugger switches back to the last program display you used, then starts executing the program at full speed.

### **L**

Switches to the low resolution graphics display with 4 lines of text at the bottom of the screen. This display mode is currently used only by ORCA/Integer BASIC.

### **N**

Switches the display to the debugger display. This command is used after using one of the other display commands to look at the program's screen. You can still step and trace while looking at the program's screen, then switch back to the debugger display using this key.

### **Q**

The debugger clicks the speaker each time you execute a line while stepping or tracing. This command turns the sound off. If the sound is already off, this command will turn it back on, again.

### **R**

The R command runs the program at full speed until the program returns from the current subroutine. The debugger does not switch back to the program's display, although you can use one of the display commands to switch to the program's display before using this command.

## ORCA/Debugger

### S

Switches to the super high resolution graphics display. This is the display normally used by all desktop programs and most Apple IIGS graphics programs.

### T

Switches to the 80 column text display. This is the display use by text programs, including those written for the ORCA or APW shell.

### X

This command is almost identical to the STEP command (i.e. pressing the space bar). The critical difference is that if the line to be executed enters a subroutine, the step command will move to the first line inside the subroutine, but the x command executes the subroutine at full speed, stopping on the next line after the one that called the subroutine.

If you are debugging a subroutine by stepping, and don't want to step through the procedures and functions the subroutine calls, use this key instead of the space bar.

## The Listing Display

Figure 3-2: The Listing Display

The listing display is an information only display. Unlike the other display areas on the debugger's screen, you never edit the source listing from within the debugger.

As you step through a program, the debugger automatically highlights the next line to be executed, scrolling the screen if necessary.

The debugger can handle programs with multiple source files. Unless you are editing the break point or memory protection displays, the debugger always shows the source file that is currently being executed. When you call a subroutine in another source file, the debugger automatically switches to the new source file, loading it from disk if necessary.

See the description of the INDENT command, earlier in this chapter, for a way to scroll the listing display left and right.



## Break Point Display Commands

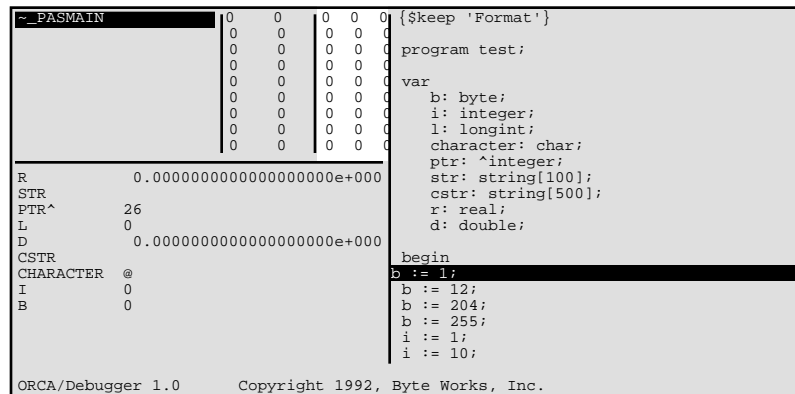


Figure 3-3: The Break Point Display

When your program triggers a break point, the program stops. If you are not already in the debugger, the debugger screen appears. The computer beeps. If you were executing the program at full speed, the debugger executes the `VERSION` command. If you were tracing a program from inside the debugger, the debugger puts you in step mode.

There are actually two different kinds of break points. Hard-coded break points are the ones you set in the source code using an editor, such as the one in the PRIZM desktop development environment. Hard-coded break points always trigger; there is no way to stop them. Hard-coded break points are generally set at the start of a short program, or just before the section of code you are interested in in a long program, and are used as an easy way to get into the debugger. Since they are triggered every time, though, it's a good idea to put them in a section of code that will only be executed once in a debugging session. Putting a source break point in the middle of a for loop, for example, isn't usually a good idea.

The memory protection display is used to set another kind or break point, the manual break point. Manual break points can be ignored by using the `OUT` command, and then reinstated with the `IN` command. You can set a manual break point in any source file, even if the debugger hasn't loaded the source file yet. You can also set a trigger value, which tells the debugger not to break until the break point has been passed a certain number of times. This is quite handy for looking at the last few steps of a loop.

There is one thing you have to be careful of when setting a break point. Not all source lines in a program are actually executed. For example, if you set a break point on the line that defines a Pascal procedure or function, or on the one that defines a C function, you won't hit the break point. The first executable line is right after the `begin` in Pascal, and at the first executable line in a C function. Basically, if the line is a declaration, any break point you set will be ignored. Break points only work when you set them on a line in the executable part of a procedure or function.

Use the BP command from the command line to start editing the break point display. While you are editing the break point display, you will see one of the fields in the display in inverse. Any of the single-key commands shown in Table 3-3 can be used. These commands are described in more detail in the sections that follow.

The three fields in the break point display are described in the section that describes the left and right arrow keys.

## ORCA/Debugger

<code>?</code>	Display a help window with this list of commands.
<code>esc</code>	Stop editing the break point display; returns to the command line.
<code>return</code>	Move to the next break point entry.
<code>left arrow</code>	Toggle between the break point line and trigger.
<code>right arrow</code>	Toggle between the break point line and trigger.
<code>tab</code>	Move to the next source file.
<code>delete</code>	Remove a break point.
<code>up arrow</code>	Move up one line in the source file. Hold down the open-apple key to move up ten lines. Hold down the option key to move up 100 lines. Hold down both the open-apple key and the option key to move up 1000 lines.
<code>down arrow</code>	Move down one line in the source file. Hold down the open-apple key to move down ten lines. Hold down the option key to move down 100 lines. Hold down both the open-apple key and the option key to move down 1000 lines.

Table 3-3: Break Point Display Commands

### `?`

The `?` key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### `esc`

The `esc` key leaves the break point display editing mode, returning you to the command line mode.

### `return`

The `return` key moves down one entry in the break point display. If you are already on the last break point, the cursor moves to the top line.

### `left arrow, right arrow`

There are three columns in the break point display. The left and right arrow keys move the cursor between the leftmost column and the middle column, allowing you to set either one.

The leftmost column is the line number where the break point is set. When you put the cursor on the break point, the debugger will automatically display the source file for that break point, highlighting the line where the break point is set.

The second column is the trigger count. When you set a break point for the first time the trigger value is set to 1.

The third column is a count of the number of times the line has actually been executed. It is set to 0 when you change a break point. (So, to reset the trigger, you could edit the break point and shift the break point up one line, then down one line.) When the value in the last column reaches the trigger count, the debugger will actually break. So, for example, you can set a break point in a loop that will execute 20 times, setting the trigger to 19. That way, the debugger will break for the next-to-last cycle through the loop.

## Chapter 3: ORCA/Debugger Command Reference

The break point line count in column three is incremented each time the break point is executed, even if you are running at full speed under program control, and can't see the break point. The break point line count is not incremented if you use the `OUT` command, which tells the debugger to ignore break points. You can reinstate break points with the `IN` command.

### **tab**

The `tab` key switches the listing window to show the next source file. If the debugger has only loaded one source file, the `tab` key is ignored.

This command lets you switch between the various source files in a multi-file program so you can set break points in a source file other than the one you are currently debugging. By using the `OPEN` command to open a source file before you edit the break point display, you can even set a break point in a source file that the debugger hasn't loaded, yet.

The only limit on the number of source files is the amount of available memory.

### **delete**

The `delete` key permanently removes a manual break point, setting all three break point fields to zero.

### **up arrow**

If the cursor is in the source code line number column, this command moves you up in the listing window. If the right or left arrow has been used to toggle to the trigger count display, this key reduces the trigger count.

If you press the up-arrow key by itself, the line number or trigger count is reduced by one. In the case of the line number, this moves you up one line in the source file.

If you hold down the open-apple key while the up-arrow key is pressed, the line number or trigger count is reduced by ten.

If you hold down the option key and press the up-arrow key, the line number is reduced by 100. The option key is ignored if you are editing the trigger count.

If you hold down both the option key and the open-apple key and press the up-arrow key, the line number is reduced by 1000. The option key is ignored if you are editing the trigger count.

Zero is the smallest value allowed in either field. If you try to set either the line number or trigger count to a value that is less than zero, the debugger sets the value to zero.

Setting the line number to zero removes the break point. You can also remove the break point using the `delete` key.

### **down arrow**

If the cursor is in the source code line number column, this command moves you down in the listing window. If the right or left arrow has been used to toggle to the trigger count display, this key increases the trigger count.

If you press the down-arrow key by itself, the line number or trigger count is incremented by one. In the case of the line number, this moves you down one line in the source file.

If you hold down the open-apple key while the down-arrow key is pressed, the line number or trigger count is incremented by ten.

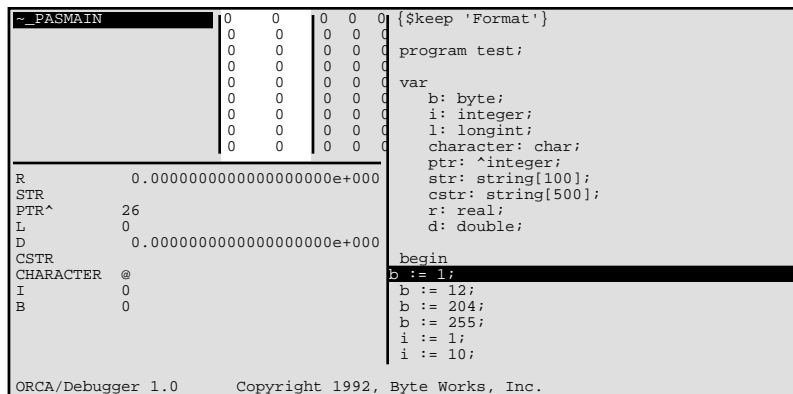
If you hold down the option key and press the down-arrow key, the line number is incremented by 100. The option key is ignored if you are editing the trigger count.

## ORCA/Debugger

If you hold down both the option key and the open-apple key and press the down-arrow key, the line number is incremented by 1000. The option key is ignored if you are editing the trigger count.

The maximum allowed trigger count is 99, and the debugger won't allow you to set the trigger count to a larger number. There is no maximum line number. If you scroll past the end of the file, the debugger will happily show you a lot of blank lines. If you scroll too far, the line number will eventually overflow, setting the line count back to a small value.

## Memory Protection Display Commands



```
~_PASMMAIN 0 0 0 0 0 ($keep 'Format')
0 0 0 0 0
0 0 0 0 0 program test;
0 0 0 0 0
0 0 0 0 0 var
0 0 0 0 0   b: byte;
0 0 0 0 0   i: integer;
0 0 0 0 0   l: longint;
0 0 0 0 0   character: char;
0 0 0 0 0   ptr: ^integer;
0 0 0 0 0   str: string[100];
0 0 0 0 0   cstr: string[500];
0 0 0 0 0   r: real;
0 0 0 0 0   d: double;

R          0.0000000000000000e+000
STR        26
PTR^       0
L          0
D          0.0000000000000000e+000
CSTR
CHARACTER @
I          0
B          0

begin
b := 1;
b := 12;
b := 204;
b := 255;
i := 1;
i := 10;

ORCA/Debugger 1.0 Copyright 1992, Byte Works, Inc.
```

Figure 3-4: The Memory Protection Display

Tracing a program takes time. When you are tracing, the debugger spends quite a lot of time updating the source window, redrawing variable values, repainting the stack frame display, and drawing the break point display as trigger values are updated. If you are tracing through a part of the program that has already been debugged, you can use memory protection to tell the debugger not to bother updating the display while the code executes, saving an enormous amount of time. It's also easier to trace through a subroutine if the subroutines it calls are protected, so you aren't constantly jumping into and out of other subroutines.

The memory protection display lets you set aside chunks of memory that will be executed at full speed. Each memory protection range is a pair of line numbers. Whenever the debugger is executing a line inside that range, it skips updating the display. Break points are still honored, so if the code triggers a break point, the program will still stop tracing.

Use the MP command to start editing the memory protection display. Once you start, you can use any of the single-keystroke commands shown in table 3-4. These commands are described in more detail in the sections that follow the table.

## Chapter 3: ORCA/Debugger Command Reference

<code>?</code>	Display a help window with this list of commands.
<code>esc</code>	Stop editing the memory protection display; returns to the command line.
<code>return</code>	Move to the next memory protection entry.
<code>left arrow</code>	Toggle between the start and end line number.
<code>right arrow</code>	Toggle between the start and end line number.
<code>tab</code>	Move to the next source file.
<code>delete</code>	Remove a memory protection range.
<code>up arrow</code>	Move up one line in the source file. Hold down the open-apple key to move up ten lines. Hold down the option key to move up 100 lines. Hold down both the open-apple key and the option key to move up 1000 lines.
<code>down arrow</code>	Move down one line in the source file. Hold down the open-apple key to move down ten lines. Hold down the option key to move down 100 lines. Hold down both the open-apple key and the option key to move down 1000 lines.

Table 3-4: The Memory Protection Commands

### `?`

The `?` key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### `esc`

The `esc` key leaves the memory protection display editing mode, returning you to the command line mode.

### `return`

The `return` key moves down one entry in the memory protection display. If you are already on the last memory protection range, the cursor moves to the top line.

### `left arrow, right arrow`

The two columns in the memory protection display are the starting line number (on the left) and the ending line number (on the right). The left and right arrow keys toggle the cursor between the two columns. While you are in either column, you can use the up and down arrow keys to change the value.

### `tab`

The `tab` key switches the listing window to show the next source file. If the debugger has only loaded one source file, the `tab` key is ignored.

This command lets you switch between the various source files in a multi-file program so you can set memory protection ranges in a source file other than the one you are currently debugging. By using the `OPEN` command to open a source file before you edit the memory protection display, you can even set a memory protection range in a source file that the debugger hasn't loaded, yet.

## ORCA/Debugger

The only limit on the number of source files is the amount of available memory.

### **delete**

The delete key permanently removes a memory protection range, setting both fields to zero.

### **up arrow**

If you press the up-arrow key by itself, the line number is reduced by one, moving the cursor up one line in the source file.

If you hold down the open-apple key while the up-arrow key is pressed, the line number is reduced by ten.

If you hold down the option key and press the up-arrow key, the line number is reduced by 100.

If you hold down both the option key and the open-apple key and press the up-arrow key, the line number is reduced by 1000.

Zero is the smallest value allowed in either field. If you try to set either line number to a value that is less than zero, the debugger sets the value to zero.

The end line number (on the right) must be greater than or equal to the start line number (on the left). If you try to set the end line number to a value smaller than the current start line number, the start line number is updated, too.

Setting both line numbers to zero removes the memory protection range. You can also remove the memory protection range using the delete key.

### **down arrow**

If you press the down-arrow key by itself, the line number is incremented by one, moving the cursor down one line in the source file.

If you hold down the open-apple key while the down-arrow key is pressed, the line number is incremented by ten.

If you hold down the option key and press the down-arrow key, the line number is incremented by 100.

If you hold down both the option key and the open-apple key and press the down-arrow key, the line number is incremented by 1000.

There is no maximum line number. If you scroll past the end of the file, the debugger will happily show you a lot of blank lines. If you scroll too far, the line number will eventually overflow, setting the line count back to a small value.

The start line number (on the left) must be less than or equal to the end line number (on the right). If you try to set the start line number to a value that is larger than the current end line number, the end line number is updated, too.



The Stack Frame display is used to change the display to show global variables, or variables from another nested subroutine. See “Stack Frame Display Commands,” later in this chapter, for details.

**Note** Pascal is a case insensitive language, while C is not. The convention that is used to support this difference is for the Pascal compiler to always report variable names as uppercase. The current version of the C compiler also converts variable names to uppercase for the debugger; this is to support a limitation in the current version of PRIZM. The ORCA/Debugger can handle mixed case variable names, and will do so automatically once the C compiler and PRIZM are updated.

## ORCA/Debugger

Table 3-5 shows the single-key commands that are available while you are viewing this display. Character commands are shown in uppercase, but the command processor is not case sensitive. You can use the equivalent lowercase letter if you prefer.

?	Display a help window with this list of commands.
esc	Stop editing the memory display; returns to the command line.
return	Move to the next variable.
down arrow	Move to the next variable.
up arrow	Move to the previous variable.
A	Display the current variable as a boolean.
B	Display the current variable as a byte.
C	Display the current variable as a character.
D	Display the current variable as a double-precision real.
E	Display the current variable as an extended-precision real.
I	Display the current variable as an integer.
L	Display the current variable as a long integer.
N	Display the current variable using the default format from the source code.
P	Display the current variable as a p-string.
R	Display the current variable as a single-precision real.
S	Display the current variable as a c-string.
X	Display the current variable as a four-byte hexadecimal number.
U	Toggle the signed/unsigned status of a byte, integer, or long integer.
^ * @ &	Any of these keys toggles the pointer mode.

Table 3-5: The Memory Display Commands

### ?

The ? key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### esc

The esc key leaves the memory display editing mode, returning you to the command line mode.

### return

### down arrow

Moves the cursor down one line. You can scroll off of the end of the list of variables, eventually filling the entire memory display with blank lines. You might even want to do this on occasion, since formatting and printing variables is one of the most time-consuming tasks the debugger has to perform during a trace, especially when any of the floating-point variable formats are used.



## Chapter 3: ORCA/Debugger Command Reference

### **up arrow**

Moves the cursor up one line. The debugger will ignore this command if the cursor is already on the first variable in the current stack frame. The memory display will scroll if the cursor is on the top line of the display and the top variable is not the first variable in the stack frame.

### **A**

Changes the format used to display the value of the currently selected variable to boolean. Boolean values are displayed as “True” or “False.”

The debugger only uses the least significant byte to determine the value of a boolean variable. In some cases, boolean values are actually two bytes long, so the displayed value may not quite match the value the program sees. You can switch the display format to byte or integer to see the actual value as either a one-byte integer or a two-byte integer.

The debugger treats zero as false and any other value as true. This matches the way the C language works. Technically, in Pascal, boolean variables always have an ordinal value of either zero or one, and any other value can cause problems.

### **B**

Changes the format used to display the value of the currently selected variable to a one-byte integer.

### **C**

Changes the format used to display the value of the currently selected variable to a character.

Non-printing characters will still be displayed as some printing character by mapping the character into the “closest” printing character. You can switch to an integer or byte format to see the actual ASCII character value.

The debugger only uses one byte to determine the value of a character. In Pascal, characters are often stored in two-byte spaces, and in C, integers can be used to store character values. In either case, comparisons may not act the way it appears they should if the most significant byte is not set to 0. For example, if a two-byte character has a value of \$1041 in RAM, the debugger will display ‘A’, which is the correct character for \$41. When you are tracing through a program, though, the comparison “ ‘A’ = ch ” (or “ ‘A’ == ch ” in C) would give a false result. To see the full, two-byte value, switch to an integer format.

### **D**

Changes the format used to display the value of the currently selected variable to a double-precision (8 byte) real number.

### **E**

Changes the format used to display the value of the currently selected variable to an extended-precision (10 byte) real number.

## ORCA/Debugger

### **I**

Changes the format used to display the value of the currently selected variable to a two-byte integer.

### **L**

Changes the format used to display the value of the currently selected variable to a four-byte integer.

### **N**

Changes the format used to display the value of the currently selected variable to the format used in the source listing. You can use this command to quickly switch a variable back to its “natural” format after manually changing the format.

### **P**

Changes the format used to display the value of the currently selected variable to a p-string. P-strings start with a length byte, and are followed by 0 to 255 characters. The debugger will print up to the length of the string, or the width of the memory display, whichever is shorter. Non-printing characters will be displayed as the “closest” printable character.

Strings are often too long to display in the memory display. You can use the RAM display to see the entire string. See “RAM Display Commands,” later in this chapter, for details.

### **R**

Changes the format used to display the value of the currently selected variable to a single-precision (4 byte) real number.

### **S**

Changes the format used to display the value of the currently selected variable to a c-string (null terminated string). The debugger will print up to the length of the string, or the width of the memory display, whichever is shorter. Non-printing characters will be displayed as the “closest” printable character.

Strings are often too long to display in the memory display. You can use the RAM display to see up to 1584 characters from the string. See “RAM Display Commands,” later in this chapter, for details.

### **U**

The three integer formats (byte, integer and long integer) can be displayed as signed or unsigned values. This command toggles between a signed and unsigned display. It does not effect any other display formats.

**X**

^ \* @ &

## Stack Frame Display Commands



You can edit the stack frame display using the `FRAME` command. All you can do is move up or down in the list of stack frames, but as you do, the variables shown in the memory window will shift to the appropriate subroutine.

## ORCA/Debugger

If you don't select a stack frame manually, the debugger automatically switches the variables window to show the topmost subroutine – the one that is currently executing. If you select a stack frame other than the top one, though, the debugger won't change the stack frame until the one you pick becomes the topmost stack frame. One convenient use for this feature is to display the global variables, then watch just those global variables as the program enters and leaves the various subroutines in the program.

Table 3-6 shows the single-key commands that are available while you are viewing this display.

<b>?</b>	Display a help window with this list of commands.
<b>esc</b>	Stop editing the stack frame display; returns to the command line.
<b>return</b>	Move down one stack frame.
<b>down arrow</b>	Move down one stack frame.
<b>up arrow</b>	Move up one stack frame.

Table 3-6: The Stack Frame Commands

### **?**

The **?** key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### **esc**

The **esc** key leaves the stack frame display mode, returning you to the command line mode.

### **return**

### **down arrow**

Moves the cursor down one stack frame. The variables window is updated to reflect the variables for the stack frame you have selected.

### **up arrow**

Moves the cursor up one stack frame. The variables window is updated to reflect the variables for the stack frame you have selected.

## RAM Display Commands

a										
+0	20	19	18	17	16	15	14	13	12	11
+10	4265	-28696	-5003	-22259	3560	30351	3564	-13438	-20734	-6043
+20	-14067	-6039	13264	26287	3560	-5943	-12275	-20694	-6011	-1406
+30	-2	8656	-1528	26746	-9638	18472	-14008	10	2768	3572
+40	-24064	6156	34	-7936	3234	8728	0	26849	-20629	-6041
+50	18445	26031	3560	-23736	18440	34	0	-1397	26746	-9638
+60	27563	-6039	13	-4850	13	0	0	0	0	0
+70	0	0	0	0	0	-1	530	-2	0	0
+80	0	0	0	0	0	0	0	0	-1	530
+90	-2	8264	-5574	23656	-3204	-29939	-21685	31482	244	-3072
+100	0	244	23040	-21030	-3302	-21176	-3304	15176	23307	-2826
+110	-25366	-5485	-27358	3562	421	773	976	22402	6145	421
+120	2153	-31488	-28671	-6654	-24061	0	-7525	-18656	-4095	-1404
+130	-4064	-14072	-4062	-14076	-12279	-14333	-4736	-18456	-4095	-1406
+140	-4064	-13852	-4062	-13856	-4087	-14116	-4736	8386	6794	2570
+150	14477	23274	25400	31233	18504	244	18432	-10833	3562	-3000
+160	-16376	244	-3072	0	674	8713	0	-28447	-32732	3342
+170	30031	8308	26223	27936	28005	29295	-2951	13	16116	-2383
+180	6924	34	-7936	-87	23807	-3204	26637	2949	-31384	-2456
+190	2	2999	-22614	-29429	-5580	13966	6378	14445	-28438	-6143
+200	2949	3462	8418	160	-18688	-26879	-4085	-14333	-2176	8386
+210	3493	901	2981	389	13997	-31254	-21235	-5580	2949	423

Figure 3-7: The RAM Display

The RAM display is a separate, full page display of the contents of memory. You can change the format used to display memory to a wide variety of number and string formats, and you can page through memory to examine large arrays or to scan for a particular area.

There are three command line commands that will bring up this display. The `var:` command displays RAM starting at the address of a particular variable. The `addr:` command displays memory starting at a particular address. The `:` command displays the RAM page as it appeared the last time you saw it, letting you switch back and forth between this display and the standard debugger display with a minimum of hassle. These commands are described in the section “The Command Line,” earlier in this chapter.

Table 3-7 shows the single-key commands that are available while you are viewing this display. Character commands are shown in uppercase, but the command processor is not case sensitive. You can use the equivalent lowercase letter if you prefer.

## ORCA/Debugger

<code>?</code>	Display a help window with this list of commands.
<code>esc</code>	Stop editing the RAM display; returns to the command line.
<code>return</code>	Move down in memory.
<code>down arrow</code>	Move down in memory.
<code>up arrow</code>	Move up in memory.
<code>A</code>	Display the current variable as a boolean.
<code>B</code>	Display the current variable as a byte.
<code>C</code>	Display the current variable as a character.
<code>D</code>	Display the current variable as a double-precision real.
<code>E</code>	Display the current variable as an extended-precision real.
<code>I</code>	Display the current variable as an integer.
<code>L</code>	Display the current variable as a long integer.
<code>P</code>	Display the current variable as a p-string.
<code>R</code>	Display the current variable as a single-precision real.
<code>S</code>	Display the current variable as a c-string.
<code>X</code>	Display the current variable as a four-byte hexadecimal number.
<code>U</code>	Toggle the signed/unsigned status of a byte, integer, or long integer.
<code>^ * @ &amp;</code>	Any of these keys toggles the pointer mode.

Table 3-7: The RAM Display Commands

### `?`

The `?` key is available from all of the displays. It shows a help window with all of the single-key commands you can use while editing the display.

### `esc`

The `esc` key leaves the RAM display mode, returning you to the command line mode.

### `return`

### `down arrow`

Moves the cursor down in memory. (“Down in memory” means to a higher memory address.)

The distance moved depends on what you are displaying and whether the open-apple key is held down when you move down in memory.

When the open-apple key is held down, you will move down 256 bytes in memory. This is independent of the display mode used, but with the exception of the string formats (which are variable length) and extended-precision floating-point numbers, all of the formats you can display have lengths which divide into 256 evenly, so you will end up on a value boundary.

If the open-apple key is not held down, the display moves down one line. The number of bytes moved will depend on the display format.

There are a few areas of memory that cannot be examined safely, namely the memory mapped I/O areas from C000 to CFFF in banks 0, 1, E0 and E1. The debugger recognizes this, displaying a `?` instead of the value.

You can also scroll to an address space that is beyond the end of the memory you actually have plugged into your computer. The debugger will display something – but it’s hard to predict what it might be.

## Chapter 3: ORCA/Debugger Command Reference

### **up arrow**

Moves the cursor up in memory. (“Up in memory” means to a lower memory address.)

The distance moved depends on what you are displaying and whether the open-apple key is held down when you move up in memory.

When the open-apple key is held down, you will move up 256 bytes in memory. This is independent of the display mode used, but with the exception of the string formats (which are variable length) and extended-precision floating-point numbers, all of the formats you can display have lengths which divide into 256 evenly, so you will end up on a value boundary.

If the open-apple key is not held down, the display moves up one line. The number of bytes moved will depend on the display format.

There are a few areas of memory that cannot be examined safely, namely the memory mapped I/O areas from C000 to CFFF in banks 0, 1, E0 and E1. The debugger recognizes this, displaying a ? instead of the value.

You can also scroll to an address space that is beyond the end of the memory you actually have plugged into your computer. The debugger will display something – but it’s hard to predict what it might be.

### **A**

Changes the format used to display RAM to boolean values. Boolean values are displayed as “True” or “False.”

The debugger assumes boolean values are one byte long, which would be correct for a packed array of boolean in Pascal, or for a boolean value or array declared as char in C. In some cases, boolean values are actually two bytes long, so the displayed value may not quite match the value the program sees. You can switch the display format to byte or integer to see the actual values as either one-byte integers or two-byte integers.

The debugger treats zero as false and any other value as true. This matches the way the C language works. Technically, in Pascal, boolean variables always have an ordinal value of either zero or one, and any other value can cause problems.

### **B**

Changes the format used to display RAM to one-byte integers.

### **C**

Changes the format used to display RAM to a series of characters.

Non-printing characters will still be displayed as some printing character by mapping the character into the “closest” printing character. You can switch to an integer or byte format to see the actual ASCII character value.

The debugger assumes characters occupy a single byte in memory, which is correct for C’s char type and for Pascal’s packed array of char and strings. In Pascal, individual character variables and unpacked arrays of char are stored as two-byte values, and in C, integers can be used to store character values. In either case, comparisons may not act the way it appears they should if the most significant byte is not set to 0, and the debugger will display an extra character for each value the program sees.

## ORCA/Debugger

### **D**

Changes the format used to display RAM to double-precision (8 byte) real numbers.

### **E**

Changes the format used to display RAM to extended-precision (10 byte) real numbers.

### **I**

Changes the format used to display RAM to two-byte integers.

### **L**

Changes the format used to display RAM to four-byte integers.

### **P**

Changes the format used to display RAM to a single p-string. P-strings start with a length byte, and are followed by 0 to 255 characters. Non-printing characters will be displayed as the “closest” printable character.

### **R**

Changes the format used to display RAM to single-precision (4 byte) real numbers.

### **S**

Changes the format used to display RAM to a single c-string (null terminated string). The debugger will print up to the length of the string, or 1584 characters, whichever is shorter. Non-printing characters will be displayed as the “closest” printable character.

### **U**

The three integer formats (byte, integer and long integer) can be displayed as signed or unsigned values. This command toggles between a signed and unsigned display. It does not effect any other display formats.

### **X**

Changes the format used to display RAM to a series of hexadecimal values. After every four bytes, a space is displayed to break the page up and make it easier to count over a specific number of bytes, but the columns formed by these blanks have no meaning at all in terms of RAM.

### **^ \* @ &**

Changes the format used to display RAM to a series of pointers. The debugger assumes each pointer occupies four bytes of memory, but it also assumes that the most significant byte is



### Chapter 3: ORCA/Debugger Command Reference

meaningless, and displays the pointers in the format `xx/xxxx`. If you need to check the most significant byte, use the hexadecimal display option.



## Chapter 4

# Debugger Utilities Reference

### DebugBreak

The `DebugBreak` utility gives you a quick and easy way to break into a program at its first executable line. This utility sends a message to the debugger, telling it to break on the next executable line of any program that uses debug code. You can then run your program from the shell, or even leave the shell to run the program from some other launcher, like the Finder. It is also possible to break into desk accessories, XCMDs, or any other program that uses debug code. You can, of course, call the utility from a script file, so that the debugger will pop up as soon as your program is built and starts to run.

There are no parameters or options for this utility. Once it is installed, just type the name at any time from the shell.

It's possible to simulate a utility style debugger using this utility and a short script. See "A Debug Script" in Chapter 2 for details.

### DebugFast and DebugNoFast

The method of debugging used by the ORCA compilers and debuggers is an intrusive method that inserts COP instructions in the executable program. There are both advantages and disadvantages to this method, but one of the disadvantages is that a program with debug code runs slower than one that does not use debug code, even if you aren't using the debugger. For some programs, the extra speed hit is severe enough to cause you to want to compile the program without debug code most of the time, adding debug code only when you want to use the debugger.

The `DebugFast` utility gives you another alternative. This utility sends a message to the debugger, telling it to patch the program. This replaces each of the COP instructions with a JMP instruction, completely bypassing both the debugger and the COP interrupt handler. For all but the tightest loops, this increases the speed of the program to almost the same speed you would get without debug code.

`DebugFast` tells the debugger to patch the RAM version of the program only. Once you decide to use the debugger, run `DebugNoFast` to tell the debugger to stop patching programs, then run the program again. Unless your program is restartable, this reloads it from disk, and the debugger will work, again. If the program is restartable you will have to remove it from memory somehow and then execute the program again. If you have no other way of purging memory, you can always reboot.

Neither utility has any options or parameters. To run the utilities, type the name from any command processor that can run ORCA or APW shell utilities.

See Appendix A for details on how the debugger works.



# Appendix A

## How the Debugger Works

### COP Vector

The ORCA compilers and debuggers use an invasive debug mechanism that depends on the compilers inserting COP instructions in the code stream. When the Apple IIGS executes a COP instruction it calls a COP handler; the debuggers insert themselves in the list of programs that are called when a COP instruction is encountered.

Several COP instructions are used. There are separate COP instructions for executing a line of source code, breaking, stepping past a line, creating symbol tables, entering and leaving subroutines, and for passing messages to the debugger. The various COP instructions are summarized in table A-1, and explained in detail below.

00	Indicates a new source code line.
01	Indicates a hard-coded break point.
02	Indicates a memory protection point.
03	Used when a new subroutine starts.
04	Marks the end of a subroutine.
05	Creates a symbol table.
06	Switches the source file.
07	Sends a message to the debugger.

Table A-1: Debugger COP Instructions

### COP 00

COP 00 indicates that a new source line has been reached and that the debugger must take appropriate action, such as updating the source listing position and variables window.

The COP instruction is followed by the line number. In assembly language, this would look like:

```
cop      00
dc       i '16'
```

### COP 01

COP 01, like COP 00, marks the start of an executable line of source code. The difference is that COP 01 also indicates that the user has marked the line as a hard-coded break point, so the debugger should break at the line.

The COP instruction is followed by the line number. In assembly language, this would look like:

## ORCA/Debugger

```
cop      01
dc       i'16'
```

### COP 02

COP 02, like COP 00, marks the start of an executable line of source code. The difference is that COP 02 marks a protected line, indicating that the debugger should not take the normal action of updating the debugger display. The only reason for putting COP 02 instructions in the code is to give the debugger a chance to override the memory protection status of a line. For example, the ORCA/Debugger allows manual break points to override these hard-coded memory protection points.

The COP instruction is followed by the line number. In assembly language, this would look like:

```
cop      02
dc       i'16'
```

### COP 03

This instruction is used right after a subroutine is called, and marks entry into the subroutine. The COP instruction is followed by the four byte address of the subroutine name, coded as a null terminated string (c-string).

```
cop      03
dc       a4'name'

...

name     dc       c'Subroutine Name',i1'0'
```

### COP 04

This instruction marks the end of a subroutine. It should appear right after the last executable line in the subroutine, but before the code that wipes out the stack frame and returns to the caller.

Debuggers will remove any symbol tables that have been created since the last COP 03 instruction.

Every COP 04 instruction must match exactly one COP 03 instruction. If the debugger encounters a COP 03 and never finds a COP 04, or encounters a COP 04 without first hitting a COP 03, it could crash or corrupt memory.

There is no operand for this instruction. In assembly language, it looks like this:

```
cop      04
```

### COP 05

COP 05 provides access to a subroutine's symbol table. It can be used after a call to vectors 3 or 6, but must be used before any calls to vectors 0, 1, and 2. The debugger's symbol table is organized as shown in Figure A-1.

## Appendix A: How the Debugger Works

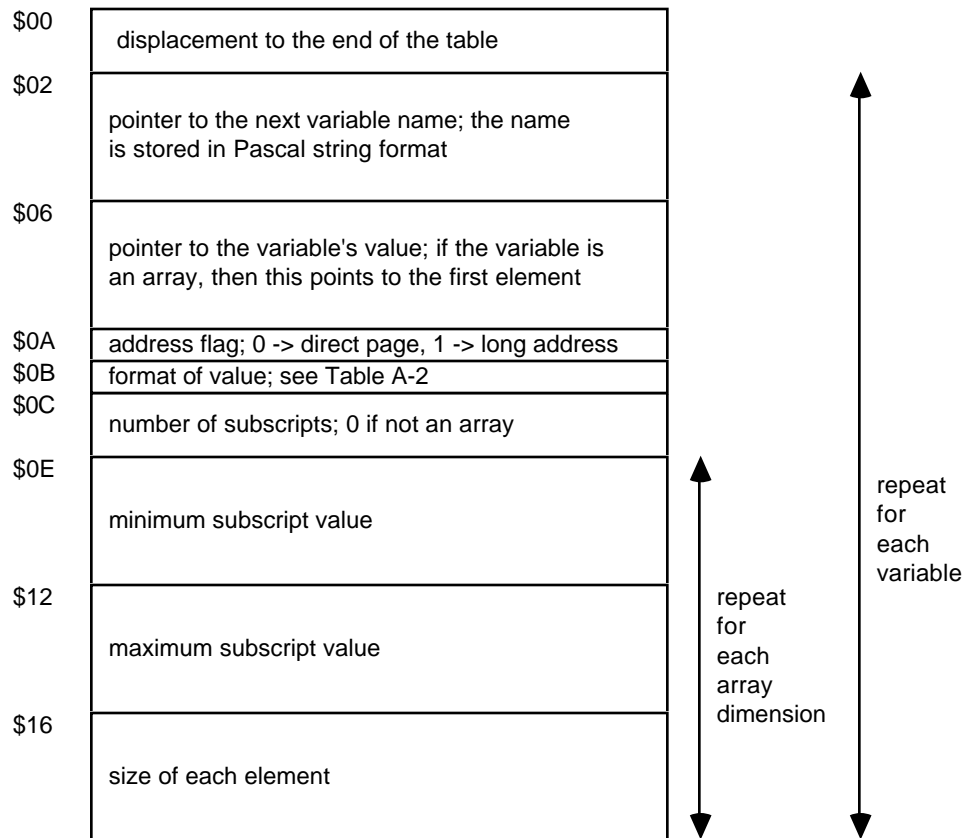


Figure A-1: Debugger Symbol Table Format

The following table shows the format used to store the variable's current value:

<u>Value</u>	<u>Format</u>
0	1-byte integer
1	2-byte integer
2	4-byte integer
3	single-precision real
4	double-precision real
5	extended-precision real
6	C-style string
7	Pascal-style string
8	character
9	boolean

Table A-2: Debugger Symbol Table Format Codes

## ORCA/Debugger

The format code indicating a pointer to any of these types of values is obtained by ORing the value's format code with \$80.

One-byte integers default to unsigned, while two-byte and four-byte integers default to signed format. ORing the format code with \$40 reverses this default, giving signed one-byte integers or unsigned four-byte integers. (The signed flag is not supported by PRIZM 1.1.3.)

The symbol table follows right after the COP 05 instruction.

## COP 06

COP 06 is used at the start of all subroutines, right after the COP 03 that marks the start of the subroutine. (You can put the COP 06 before or after any COP 05, so long as it comes before any COP 00, COP 01 or COP 02 instructions). This instruction flags the source file for the subroutine, giving the debugger a chance to switch to the correct source file if it is not already being displayed. You can also imbed other COP 06 instructions inside of the subroutine if the subroutine spans several source files.

The COP 06 instruction is followed by the four-byte address of the full path name of the source file. The path name is given as a C-string. The ORCA/Debugger supports path names up to 255 characters long, and allows either / or : characters as separators. Here's what the instruction might look like in assembly language:

```
cop      06
dc       a4 'name '
...
name     dc      c '/hd/programs/source.pas',i1'0'
```

## COP 07

COP 07 is used to send messages to the debugger. The first four bytes following the COP 07 have a fixed format, but the remaining bytes vary from message to message.

The two bytes right after the COP 07 instruction are the total length of the debugger message, in bytes. This will always be at least 4. The next two bytes are the message number. The message number can be followed by more bytes.

Three messages are currently defined and supported by ORCA/Debugger. None uses any optional fields, so the length word should be four for all three of these messages.

Message 0 tells the debugger to start patching all debugger COP instructions with JMP instructions. This is the message sent by the DebugFast utility. This message must be sent before a program starts to execute – sending this message after a program with debug code starts, but before it finishes, can cause memory corruption or crashes.

Message 1 tells the debugger to stop patching COP instructions, reversing the effect of message 0. The DebugNoFast utility sends this message.

Message 2 tells the debugger to treat the next COP 00 as if it were a COP 01. The DebugBreak utility sends this message.



## Index

### special characters

: command 18  
? command 9, 18  
"var: command 8, 15  
"var=val command 15

### A

addr: command 19  
address format 19  
APW 24  
auto-go 11, 15, 20  
    see also memory protection

### B

booleans 18, 33, 39  
BP command 19  
break point display 4, 25  
break point trigger 26  
break points 9, 11, 13, 19, 20, 21, 25-28, 45  
    see also hard-coded break points  
bytes 33, 39

### C

c-strings 17, 34, 40  
case sensitivity 31  
characters 17, 33, 39  
CLR command 19  
command line 14  
command line display 5

### D

debug code 10, 11, 12, 43, 48  
DEBUG utility 11  
DebugBreak utility 3, 11, 13, 43, 48  
DebugFast utility 3, 12, 43, 48  
DebugNoFast utility 3, 43, 48  
desktop 24  
double-precision 16

### E

entering the debugger 4, 10-12, **13-14**  
event loops 13

Event Manager 13  
execution speed 12  
extended precision 16

### F

four-finger salute 11, 13, 20  
FRAME command 19

### H

hard-coded break points 9, 11, 20, 25, 45, 46  
    see also break points  
help 5, 18, 22, 26, 29, 32, 36, 38  
hexadecimal values 35, 40

### I

IN command 19, 27  
INDENT command 20  
installation 3  
integers 16, 33, 34, 39, 40

### J

jump command 5, 20

### K

keyboard 13

### L

leaving the debugger 14, 20  
listing display 5, 20, 24, 27, 29  
long integers 34, 40  
low resolution graphics 5

### M

manual break points 25  
MEM command 20  
memory display 4, 6-8, 31  
memory protect display 4  
memory protection 10, 28-30, 46  
MP command 20  
multiple files 27, 29

## ORCA/Debugger

### O

OPEN command 9, 20, 27, 29  
ORCA 24  
ORCA/C 11  
OUT command 21, 27

### P

p-strings 17, 34, 40  
pointers 18, 35, 40  
PRIZM 10, 11, 48

### R

RAM display 18, 19  
readln 14  
real numbers 16, 33, 34, 40

### S

scanf 14  
scripts 11  
shell 10, 24  
single-precision 16  
sound 23  
source break points  
    see hard-coded break points  
source code 20, 24, 27, 29  
speed 12  
stack frame display 4, 35  
stack frames 6, 8, 19, 31  
step 22  
step command 5, 21  
step/trace mode 5, 6, 14, 21, 24  
strings 17, 34, 40  
subroutines 6, 23, 24, 46  
    see also stack frames  
super high resolution graphics 6, 24  
symbol table 46

### T

text display 24  
text screen 5, 6  
trace 22  
trace command 5, 21

### U

unsigned integers 31, 34, 40  
utility style debugger 11

### V

variable name case 31  
variables 6-8, 18, 19, 20, 31, 46  
    setting 15  
VERSION command 14, 21