ORCA Disassembler™

for the Apple® IIGS

Paul Elseth

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

Limited Warranty - Subject to the below stated limitations, Byte Works Inc. hereby warrants that the program 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.

Table of Contents

Chapter 1
Introduction

Features

ORCA/Disassembler is a full featured disassembler for the Apple IIGS. Its features include:

- disassembly of ROM, BIN, SYS, and OMF files.
- two versions – desktop and text.
- recognition of toolbox macros and GS/OS calls.
- a script command language.
- automatic generation of ASM65816 USING, APPEND, and GEQU directives.
- multiple direct pages.
- tracking of register/memory width.
- handling of addresses which are offsets into data areas.
- full-screen scrolling disassembly.

What You Should Have Received

The ORCA/Disassembler package consists of one 3.5-inch disk, this manual, and a warranty registration card. Be sure to return the registration card – it is our proof that you purchased the product, and will ensure that you are added to our database to receive information about future product updates.

The ORCA/Disassembler disk contains two versions of the disassembler, a stand-alone desktop version, and a text-based version. The desktop version is more powerful, easier to learn, and uses windows and pull-down menus. The text version is designed for those who prefer a faster, command-driven interface.

About This Manual

This manual is written assuming that you will be using the desktop version of ORCA/Disassembler. Chapter 5 discusses the differences between the desktop disassembler and the text-based version.

Chapter 2 is designed to get you up and going on ORCA/Disassembler quickly. Chapters 3 and 4 give more detail about the options and commands available.

System Requirements

Software Requirements

The desktop version of ORCA/Disassembler requires GS/OS system software 5.0 or later. The text-based version of ORCA/Disassembler requires either the APW or ORCA/M shell, and GS/OS system software version 4.0 or later. To use the disassembler effectively, you must have a basic understanding of the ORCA/APW assembler.

Hardware Requirements

The minimum hardware configuration required to run ORCA/Disassembler is:

- An Apple IIGS computer, or an Apple //e computer with an installed Apple IIGS upgrade, with 256K bytes of RAM.

- The text-based version of ORCA/Disassembler requires an additional 256K bytes of RAM, for a total of 512K bytes of RAM.

- The desktop version of ORCA/Disassembler requires an additional 544K bytes of RAM, for a total of 1024K bytes of RAM.

- One 3.5-inch disk drive.

Recommended Hardware

The more memory that you have, the larger the programs you will be able to disassemble.

Installation

First make a back-up copy of the /ORCA.DISASM disk!! NEVER work from original disks! ORCA/Disassembler is not copy-protected; you can use any disk copying program to make your working copy of the disk.

If you will be using the desktop version of ORCA/Disassembler you don't need to do anything special to install the disassembler. You can move the disassembler and its associated files to a directory on another disk. The files that you will need are all located on the /ORCA.DISASM disk; they are named DESKTOP.DISASM (the desktop version of the disassembler), DISASM.SCRIPTS, DISASM.CONFIG, and DISASM.DATA. Note that the DESKTOP.DISASM file has a resource fork! Be sure to use a program launcher which supports resource forks, such as the Finder, to copy the file. This caution about the resource fork does not apply to block-by-block disk copying utilities. If you use the ORCA shell to copy the files, be sure the shell is version 2.0 or later. Earlier versions of the shell do not copy the resource fork of extended files.

The desktop disassembler, as shipped, is an S16 file. When you execute an S16 file from the shell, the shell shuts down. When the S16 file is finished, it reboots the shell. If you will be launching the disassembler from ORCA or APW, you may want to change it to an EXE file. *(Note: You must version 2.0 or later of the ORCA or APW shells to use the desktop version of the*

disassembler as an EXE file!) You can convert the DESKTOP.DISASM file to an EXE file with the FILETYPE command of ORCA/APW:

```
filetype desktop.disasm exe
```

If you will be using the text-based version of ORCA/Disassembler, you will probably want to install the disassembler as an APW or ORCA/M utility. To do so, follow these steps:

1. Edit the file named SYSCMND, located in the SYSTEM directory of your APW or ORCA/M system disk.

2. As you can see, SYSCMND is alphabetized by command name. Create a blank line in the file where "DISASM" will appear. Type the name of the utility, DISASM, in the first column of the table. This is the name of the text-based disassembler. In the second column of the table, enter a "*U," for utility (the "*" means that this utility is restartable). Leave the third column blank. The final column can be used for a comment.

3. Save the updated SYSCMND file.

4. Copy the DISASM, DISASM.DATA, DISASM.CONFIG, and DISASM.SCRIPTS files from the /ORCA.DISASM disk to the UTILITIES folder of your APW or ORCA/M disk.

5. Copy the file named DISASM, located in the HELP folder of your /ORCA.DISASM disk into the UTILITIES/HELP folder of your APW or ORCA/M system.

Chapter 2
Exploring The System

In this chapter we will briefly look at ORCA/Disassembler as a whole. You should already have read Chapter Three of the *Apple IIGS Owner's Guide*, which came with your computer, and be familiar with basic desktop terminology, as well as with mouse operations.

This manual also assumes that you are familiar with the hierarchical directory structure used by GS/OS, and that you understand such terms as prefix, directory, volume, root directory, and subdirectory. Subdirectories and folders are used interchangeably in this manual. You should also understand the terms filename and pathname.

Running ORCA/Disassembler

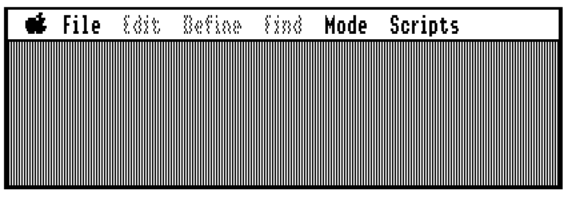
The disassembler can be run by launching it from any program launcher, including APW and ORCA/M. If the program launcher you are using is the Finder, for example, you would run the disassembler by double-clicking on its icon on the /ORCA.DISASM disk. To run it from APW or ORCA/M, you would first set your prefix to the /ORCA.DISASM disk, and then enter the name of the disassembler program as a command:

```
#prefix /ORCA.DISASM
#desktop.disasm
```

When ORCA/Disassembler is executed, it loads the files DISASM.SCIPTS, the file containing the disassembler script commands; DISASM.CONFIG, the file containing mode defaults; and DISASM.DATA, the file containing toolbox, shell, and operating system call names.

The Menu Bar

First, launch ORCA/Disassembler. You should see the ORCA/Disassembler menu bar, pictured below:



Get a feel for the program by pulling down each of the menus. The `Apple`, `File`, and `Edit` menus are standard Apple desktop menus. You will use the `File` menu to perform basic file operations such as opening, printing, and saving. The `Edit` menu contains the standard editing commands Cut, Copy, and Paste. The `Define` menu contains commands for defining labels, comments, data areas, and so on. The `Find` menu is used to locate specific data in your disassembled file, and to move quickly to exact locations within the disassembly. The `Mode` menu is used to control the way in which the disassembly is displayed. The `Scripts` menu provides your interface with the disassembler's script command language. The last menu, the `Segments` menu, is activated once you have opened a file to disassemble. It provides information about the segments in the disassembled file.

Menu Items

You may have noticed that some of the menu commands have an open-Apple (Ó) followed by a character. These two characters together comprise what is called a command equivalent (or keyboard equivalent). To execute commands which have keyboard equivalents, you can either pull down the menu and select the command, or you can hold down the open-Apple (Ó) key while simultaneously pressing the character given in the menu. For example, to issue the `Open...` command (available under the `File` menu) from the keyboard, you would hold down the open-Apple key (Ó) and then press the key for the letter o. Note that you can press either a capital O or a lower-case o.

Many of the menu commands have three dots (...) after the command name. The dots mean that the command will bring up a dialog box which you will use to execute the command. All dialog boxes used in ORCA/Disassembler are discussed in detail in Chapter 3.

A dimmed menu item means that the command is not selectable at this time.

Apple Menu

Let's look at the `Apple` menu for a moment. When you pull down the `Apple` menu, you will see the item "`About ORCA/DISASSEMBLER...`" with a line underneath. Most desktop programs featuring an `About` item use it to show the version number of the program and the name of the program's author. Any items below the line are desk accessories; these can be accessed at any time during the operation of ORCA/Disassembler. You can also add new desk accessories by placing them in the DESK.ACCS folder of the boot disk's SYSTEM folder. Feel free to select the `About` item and any desk accessories.

Selecting

"Selecting" refers to the process of clicking the mouse on text, or using the mouse to drag over text. The selected text is highlighted. There are a variety of reasons for selecting text in the disassembler, including:

- The selection will determine the default addresses for many of ORCA/Disassembler's operations. For example, if address $0010 is selected, then if you select the `Define Label...` command, the label address will default to $0010, and the label name will default to the label currently defined at $0010, if a label has been defined for this address.

- Any text that is selected when you issue the `Print` command will be printed. This allows you to print only a portion of the current segment.

In normal (non-hex) mode, selection is always by line. In hex mode, selection is always by bytes.

Selection by Dragging

To select a line, click on it with the mouse. You can select multiple lines by clicking on a line which starts the block of text to select, and then dragging the mouse while holding down the mouse button. Release the mouse button when you are done selecting.

In hex mode, you select a byte by clicking on it. You can select a range of bytes by dragging the mouse while holding down the mouse button. Release the mouse button when you are done selecting.

If you move the mouse off of the top or bottom of the window, the window will start to scroll. This allows you to select more than can be seen in the window at one time.

Deselecting

Text is deselected by clicking the mouse in the left margin of the window, between the edge of the window and start of the disassembly listing.

Selecting the Entire Segment

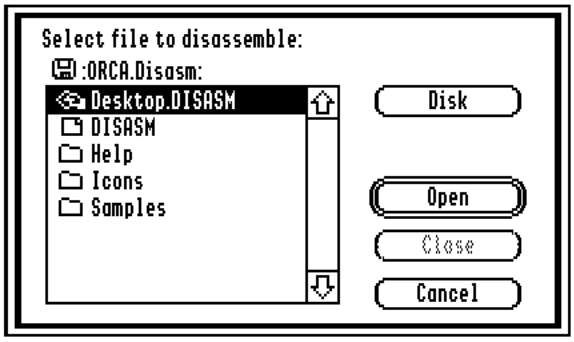
The `Select All` command, located in the `Edit` menu, selects the entire segment.

Extending a Selection

Extending a selection is a method that is generally used to select large ranges, although it can be used to modify the extent of an existing selection. First, place the cursor at one end of the range you want to select. Now go to the point in the text where the selection is to end. (You can use scrolling or the `Goto` command, available in the `Find` menu.) Hold down the shift key, and click the mouse button or continue selecting. The entire block, from the original selection to the new position, is selected.

Disassembling Your First Program

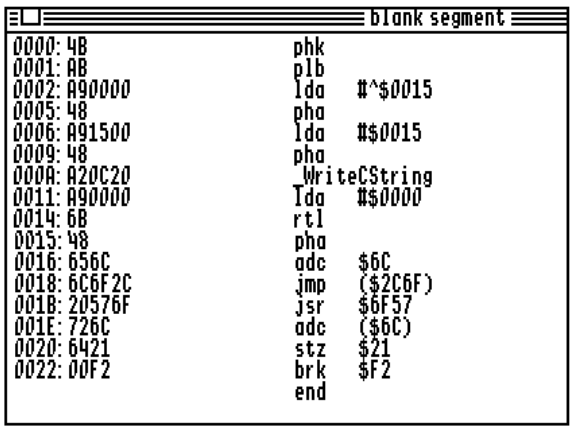
To demonstrate how to disassemble a program, let's work with the sample programs included on the /ORCA.DISASM disk. Start by pulling down the `File` menu and selecting the `Open...` command. `Open...` brings up a dialog box similar to that pictured below:



Notice the icons in the `Open` dialog. Those which look like a folder represent a subdirectory. In order to access the program, we first need to open the SAMPLES directory. Click the mouse button on SAMPLES, and then select the `Open` button. We can also open the SAMPLES directory by double-clicking the mouse button on SAMPLES, or by selecting SAMPLES with one click, and then pressing the RETURN key.

As you can see, the `Open` button is outlined, indicating that this is the default button. The `Open` dialog should now be displaying the list of files for the SAMPLES directory. Select the file named WRITELN with the mouse, and then open it by clicking on the `Open` button or by pressing the RETURN key.

The disassembler creates a new window named "blank segment" containing the disassembled code from the WRITELN file. Let's open the window to its full size; you can do this by clicking the mouse on the zoom box, located in the upper right-hand corner of the window. The contents of the window should look like the picture below:

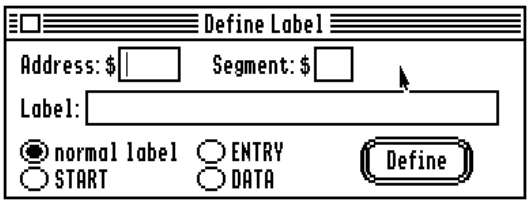


The four-digit number on the far left is the offset from the beginning of the segment to instruction on that line. The PHK instruction, for example, starts 0 bytes from the beginning, while the RTL instruction is located $14 bytes from the start of the segment. You will use these offsets as the addresses for things you'll define in the course of converting the disassembly into an assembly-language source file.

Following the offset is a colon and then some more hexadecimal digits. These digits give the machine code for the line.

After the machine code is the assembly language source code derived from the machine code. The first field is the opcode field, and the second field is for the operand. Some instructions do not have operands, so this field can be blank.

The first change we'll want to make to the disassembly in order to convert it to source code is to add a START directive. (The START directive is used in the ORCA/M and APW assemblers to mark the beginning of a new code subroutine.) Pull down the `Define` menu and select the `Define Label...` command. This causes a dialog box like the one pictured below to appear:



In the `Address` box, enter 0, the address for the START directive. In the `Label` box, enter WriteLn, the name of the label to attach to this segment. Now click the mouse button on the radio button next to `START`. Once you've filled in the dialog box correctly, click the mouse button on the `Define` button (or press the RETURN key since the `Define` button is the default button). We will not worry about the Segment box now; we'll see how it is used when we move on to our next sample. The `Define Label` dialog stays on the screen, just in case you want to define several labels. Close the dialog by clicking on the close box.

Let's look at the lines in the disassembly which deal with the value of $0015:

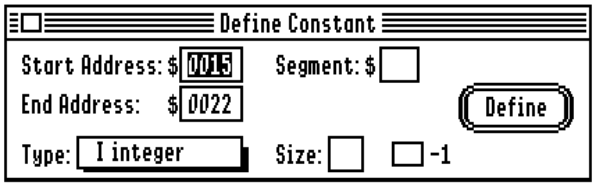
```
0002:A90000          lda    #^$0015
0005:48              pha
0006:A91500          lda    #$0015
0009:48              pha
000A:A20C20          _WriteCString

                     ...

0014:6B              RTL
0015:48              PHA
```

The $0015 is probably a label. We see that the $0015 appears in the disassembly in the offset column at the beginning of the line immediately following the RTL instruction. We conclude that $0015 must be the label for some data, that the LDA instructions involving $0015

are taking the address of the label, and that the strange instructions occurring between the RTL and END must be data.

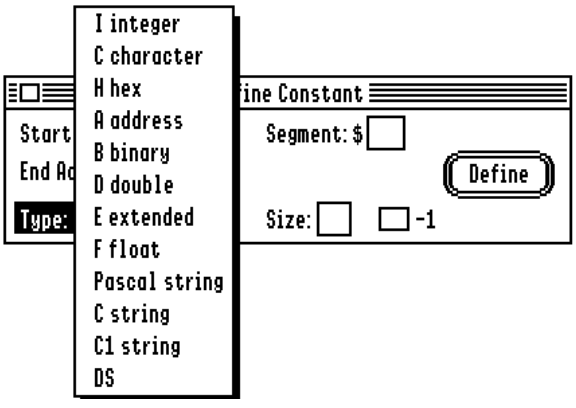
Pull down the `Define` menu and select the final item, `Generate Labels`. After the command is executed, we see that the disassembler agrees with our observations about $0015. The `Generate Labels` command automatically generates labels for the current segment. As you can see, the label names begin with the letter L, followed by four hexadecimal digits corresponding to the location of the label in the current segment.

One last step needs to be taken to convert our disassembly into a valid assembly-language source file. Use the mouse button to select the lines between the RTL and END. Now pull down the `Define` menu and select the `Define Constant...` command. It brings up a dialog box like the one pictured below:



Notice how the address boxes are already filled in for us. That's because we had selected some lines prior to issuing the `Define Constant` command. When we were giving the first line of the disassembly a START label, we could have clicked on the first line, and then issued the `Define Label` command to cause the disassembler to automatically fill in the label's address. We will need to adjust the `End Address` in the dialog. It is filled in with $0022, the address of the first byte of the line. We want to include the entire line, so we look at the machine code for the line. We see that the line is two bytes long, so type $0023 in the `End Address` box.

Beneath the address boxes is a pop-up menu entitled `Type`. Click on `Type`, then keep the mouse button depressed, to see the types of data that we can define:

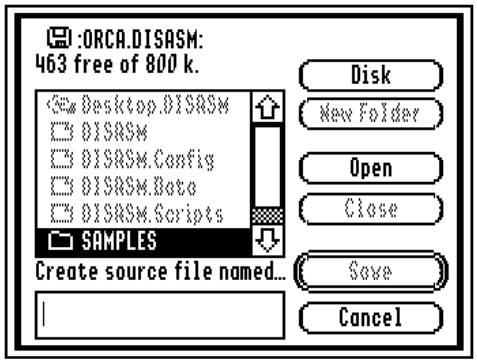


When the pop-up first comes up, you'll see a blank area and then some menu items, followed by an arrow pointing down. The arrow indicates there are more items available. Use the

mouse to drag downward on the arrow, which causes the pop-up to scroll upward, filling in the blank area.

In the disassembly, the program is pushing the address of some data, and then calling the Quick Draw II function WriteCString. The _WriteCString instruction in the disassembly is a macro call. Both APW and ORCA/M use the same convention in naming macros that call the Apple IIGS toolbox: they use an underscore followed by the name of the tool call as it appears in the *Toolbox Reference* manuals. The data is probably a C string (a null-terminated string), so we select `C string` from the `Type` pop-up, and then click the `Define` button. After the command executes, we see that the disassembler has "cleaned up" the strange instructions by using DC directives to convert the lines to character constants followed by a hexadecimal constant.

Our disassembly is now ready to be converted to a source file. Select the `Create Source File...` command from the `File` menu. It brings up a dialog box like the one pictured below:



Use the `Disk` button to move to the disk where the source file is to be saved. If you would like to create a subdirectory for the source file, enter the name of the new subdirectory in the `Create source file named...` box, click on the `New Folder` button, and then click on the `Open` button to open the new folder. Now enter the name of the source file in the `Create source file named...` box. For our example, enter the name WRITELN.ASM. Finally, click on the `Save` button.

At this time it would be a good idea to see what our new source file looks like. Exit the disassembler program by selecting the `Quit` command from the `File` menu. You will be presented with an alert box, similar to the one below:

We will be using WRITELN's template in our next example, so let's save its template file. Do this by clicking on the `Save` button. This causes the standard `Save` dialog box to appear. Move to the disk and subdirectory where the source file has been saved, then enter the name WRITELN.TEMP in the `Save template as...` box. Now click on the `Save` button.

Let's examine the source file we've just created. (You can display the file, print it, or edit it at this point to examine its contents.) The file should look like the one below:

```
*  ORIGINALLY $0000
WriteLn  start
         phk
         plb
         lda   #^L0015
         pha
         lda   #L0015
         pha
         _WriteCString
         lda   #$0000
         rtl
L0015    dc    c'Hello, World!'
         dc    h'00'
         end
```

As you can see, all we need to do to prepare this file for assembly is to add an MCOPY directive to the top of the file to access the _WriteCString macro, and then run the source file through the MACGEN utility (available with both APW and ORCA/M) to generate its macro file.

A Second Look At The Disassembler

Templates

Let's briefly return to the disassembler to learn about templates and the `Mode` menu. First, launch the disassembler. Next, open the file WRITELN located in the SAMPLES directory. Now select the `Load Template...` command from the `File` menu. This brings up a standard `Open` dialog box. Use the `Disk` and `Open` buttons as needed to open the template file WRITELN.TEMP that we saved earlier.

Note what happens to our disassembly after we load the template file! The labels and directives are placed into the disassembly in the same way we originally defined them. The template file contains all of the changes you made to the disassembled file from the time the file to be disassembled was opened to the time the template file was created. You can use templates to record various changes to the disassembly, and then recall the changes by loading the different templates.

Modes

We will conclude this part of our tutorial with a look at the Mode menu. Pull down the Mode menu and select the View hex command. The display of the disassembled file is changed to a series of hexadecimal values. Each line contains a four-digit hexadecimal value giving the offset of the beginning of the line from the start of the segment. After the offset is a colon and then sixteen two-digit hexadecimal values. After the hexadecimal values are a few spaces and then some characters which represent the ASCII characters of the two-digit hexadecimal numbers. This form of disassembly is called a "hex dump."

Select the View hex command again to change the display back to assembly-language source code. Now select the Toolbox macros command from the Mode menu. We see that the _WriteCString macro call has been replaced with lines similar to these:

```
000A:A20C20         LDX  #$200C
000D:220000E1       JSL  >ToolBox      WriteCString
```

These lines are what are contained in the _WriteCString macro: the X register is loaded with the function and tool number, and then a JSL to the toolbox entry vector is done.

A look at the machine code for the line containing the JSL shows that the ToolBox label should be equated with hexadecimal E10000. (Bytes are stored into memory on the Apple IIGS as least significant to most significant.) When you create a source file for a disassembled segment containing a label like ToolBox, the disassembler automatically generates GEQU directives for the labels it uses.

Disassembling A Larger Program

We will now disassemble a larger program, introducing some other features of the disassembler. Pull down the File menu and select the Open command. Open the Samples folder, and then open the file named Sample.

The disassembler creates a new window named "blank segment" containing the disassembled code from the first segment of the Sample file. Pull down the Segments menu. Notice that there are three items below the dividing line – ~ExpressLoad (dimmed), *blank segment*, and SEGMENT3. These items correspond to the three segments in the Sample program. The first segment, ~ExpressLoad, is not actually part of the program – it is information used by the system, and will not always be found in a given program. Blank segment is not actually the name of the second segment. The actual segment name is “ ” (10 spaces). The disassembler refers to segments with this name as the blank segment.

You can select either of the other two undimmed segments by selecting the appropriate item from the Segments menu. Note that there is a check mark by the “blank segment” item; this indicates that the “blank segment” is the currently selected segment. The first segment in a program is automatically opened when the file is loaded. Now click on the zoom box of the “blank segment” window to expand the window to full size. The contents of the window should look like the picture below:

```
0000: 4B              phk
0001: AB              plb
0002: 22000003        jsl   >$03/0000
0006: B003            bcs   $000B
0008: 201B00          jsr   $001B
000B: 203502          jsr   $0235
000E: 22A800E1        jsl   >GSOS                    Quit
0012: 2920            dc    i2'$2029'
0014: 19000000        dc    i4'$0019'
0018: 6B              rtl
0019: 0000            brk   $00
001B: A204CA          _InitCursor
0022: 20B700          jsr   $00B7
0025: 9001            bcc   $0028
0027: 60              rts
0028: 9C0502          stz   $0205
002B: AD0502          lda   $0205
002E: D020            bne   $0050
0030: 48              pha
0031: F4FFFF          pea   $FFFF
```

The first thing we'll do is to create a START directive and name the first routine in the program. Select the first line in the segment by clicking anywhere on the line. The selected line is used as the target for many of the disassembler's commands. Now select `Define Label` from the `Define` menu (or press ♁D).

Notice that the `Address` box is already set to "0," because we selected the line corresponding to address $0000 in the disassembly window. We can leave the `Segment` box empty for now, since the default is the current segment. Since the insertion point is already in the `Label` box, type "Sample" as the label. Now click the `START` radio button – this will cause our label to be defined as a START directive. Next click the `Define` button (or press the RETURN key). Finally, click on the "blank segment" window again to bring this window in front of the `Define Label` window.

There is another, shortcut way to define a label. Double-click on the Sample label we just defined. Note that the line changes so that only the label is highlighted. You may now edit the label using the standard text editing commands. Press the RETURN key after editing it to define the label. You can also add new local labels to a line by double-clicking in the label area of the line. Now select the first line again by clicking on it once. Press OPTION-S. Press OPTION-S again. The OPTION-S command adds and removes a START directive on the selected line.

Note that the third line of the program is "jsl >$03/0000". The '/' in the line indicates that the address is a long reference to a different segment, in this case the third segment ("SEGMENT3"), and not a reference to bank 3 of memory. Long references to absolute memory locations do not contain the '/' separating the bank from the address. For example, a long call to bank $E0, address $0100, would appear as "jsl >$E00100."

Since this is a jsl to another segment, lets look at that segment now. Pull down the `Segments` menu and select "SEGMENT3." This creates another window, which contains the disassembly of the third segment in the Sample program:

```
0000: 7B              tdc
0001: 8DFB01          sta   |$02/01FB
0004: A20102          _TLStartUp
0008: 48              pha
0009: A20202          _MMStartUp
000C: 68              pla
000D: 8DFD01          sta   |$02/01FD
0010: 9C0102          stz   |$02/0201
0013: 9CFF01          stz   |$02/01FF
0016: 48              pha
0017: 48              pha
0018: ADFD01          lda   |$02/01FD
001B: 48              pha
001C: F40200          pea   $0002
001F: F40000          pea   $0000
0022: F40100          pea   $0001
0025: A20118          _StartUpTools
0028: FA              plx
0029: 7A              ply
002A: 9001            bcc   $002D
```

As a shortcut, you may use OPTION-(segment number) to select segments. In this case, we could use OPTION-3 to select the third segment.

We won't assign a label to $03/0000 now, since we don't yet know what it does. The first instructions are "tdc, sta |$02/01FB." These instructions save the value of the direct page register at address $01FB in segment 2. The '|' indicates that the instruction uses the short (2-byte), rather than long (3-byte) addressing mode. Let's define a label called “DPage” at $01FB in segment 2. To do this, select `Define Label` again. Enter “01FB” as the starting address, “2” as the segment, and “DPage” as the label. Make sure the `normal label` radio button is selected, then click the `Define` button. Returning to the disassembly, we see that the second instruction changes to “sta |DPAGE.”

Let's see what effect that had in the segment where DPage is defined. First select `Save Location` from the `Find` menu to place a “bookmark” in our current segment so we can return here quickly. Then select segment “blank segment” from the `Segments` menu (or press OPTION-2). To find DPage, select `Goto` from the `Find` menu (ÓG). Enter “01FB” as the address, then click `OK`. The contents of the window scroll to show the disassembly at $01FB at the top of the window. As you can see, address $01FB is now labelled “DPage:”

```
01FB: 0000        DPage    brk   $00
01FD: 0000                 brk   $00
01FF: 0000                 brk   $00
0201: 0000                 brk   $00
0203: 0000                 brk   $00
0205: 0000                 brk   $00
0207: 0000                 brk   $00
0209: 0000                 brk   $00
020B: 0000                 brk   $00
020D: 0000                 brk   $00
020F: 0000                 brk   $00
0211: 0000                 brk   $00
0213: 0000                 brk   $00
0215: 0000                 brk   $00
0217: 0000                 brk   $00
0219: 0000                 brk   $00
021B: FFBF1F00             sbc   >$001FBF,x
021F: 0000                 brk   $00
0221: 0000                 brk   $00
0223: 0000                 brk   $00
```

Note that the instruction at DPage is “brk $00.” Since we know that DPAGE is actually data, let's change the instruction to a constant definition. Click on the top line to select it, then select `Define Constant` from the `Define` menu (ÓK).

The `Start Address` and `End Address` are preset to the selection, $01FB. The segment will default to the current segment, so we'll leave it blank. The `Size` box is used to define the size of the ‘I’ and ‘A’ integer constants, but since it defaults to 2, we'll leave it blank also. Select the `I integer` item from the `Type` pop-up menu. The remaining option is the `-1` check box. This is to generate the “DC A‘Label-1’” entries that are often used in jump tables – see the description of `Define Constants…` in Chapter 3 (Menu Commands) for more information about the -1 check box. At this point, we're all set, so click the `Define` button. The top line in the window will change to “DPage DC I2‘$0000’.”

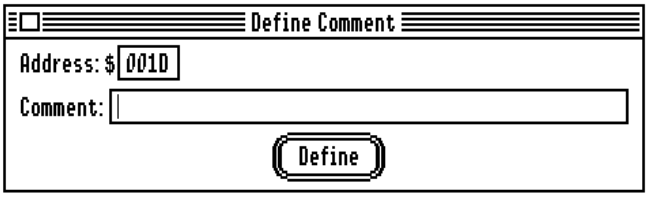
There is a shortcut method for defining commonly used constants. We could have simply selected the line at address $01FB, then pressed OPTION-W (for “word”), and the disassembler would have defined a 2-byte integer at the selected address. Other similar shortcuts are given in chapter 3. Now that our constant definition is complete, let's return to where we were in segment 3 by using the `Restore Location` command from the `Find` menu.

The next instruction in segment 3 (at address $0002) is “_TLStartup.” “_TLStartup” is a predefined toolbox macro used to start the Tool Locator. (All built-in toolbox macros and machine globals are located in the file DISASM.DATA. You may edit this file with any text editor to see what items are predefined, and to change or add additional items to the list. Further information about DISASM.DATA can be found in Appendix B.) TLStartup takes no arguments, and returns no results.

The next instruction sequence is “pha, _MMStartup, pla, sta $02/01FD.” “_MMStartup” is the macro to start the Memory Manager. It returns an integer value, the application's Memory Manager user ID, so the “sta $02/01FD” must be saving the application's user ID. Define the label “userID” at address $01FD in segment 2, and make it a two-byte integer constant. Note that you don't have to switch back to segment 2 to do this – just use the `Segment` parameters in the `Define Label` and `Define Constant` dialogs.

The next two instructions store zero to two addresses. Since we don't yet know what that these two addresses signify, we'll continue on.

Notice a line that reads "_StartUpTools" coming up. This tool call loads and starts tools sets, takes three parameters, and returns a long (4-byte) result. The "pha, pha" instructions reserve stack space for the result, and the instructions following them push the parameters required by the tool call. To make things easier, let's comment these lines. Select address $001D then select the Define Comment (;) command from the Define menu. Its dialog looks appears below:



Fill in "(long result)" for the comment, then click the Define button. We'll use the shortcut method to define the other comments: click on the segment window to bring it to the front, then double-click on line $001F, right beneath the comment we just defined for line $001D. You should see the blinking insertion point on that line. Simply type "userID" then press the RETURN key. Similarly, comment $0023 as "startStopDesc," and $0026 as "startStopRecRef." (These comments are the names that the *ToolBox Reference* manual uses for the parameters.)

Since startStopRecRef is a long, it takes two "pea" instructions to push the value. A startStopDesc value of $0002 tells us that the startStopRecRef is a resource ID, so that the startStopRecord is not here in the program itself, but rather resides in a resource. You can use Derez® or a similar tool to look at the resources in a file.

Next the program pulls the long result, the startStopRecRefRet(!) It then checks if an error was returned by the Tool Locator, exiting the subroutine if an error condition exists. If not, it saves the result at $02/01FF:

```
001D: 48          pha                        (long result)
001E: 48          pha
001F: ADFD01      lda   |userID               user ID
0022: 48          pha
0023: F40200      pea   $0002                startStopDesc
0026: F40000      pea   $0000                startStopRecRef
0029: F40100      pea   $0001
002C: A20118      _StartUpTools
0033: FA          plx
0034: 7A          ply
0035: 9001        bcc   $0038
0037: 6B          rtl
0038: 8C0102      sty   |$02/0201
003B: 8EFF01      stx   |$02/01FF
003E: 48          pha
003F: 48          pha
0040: F40200      pea   $0002
0043: F40000      pea   $0000
0046: F40100      pea   $0001
0049: F40000      pea   $0000
```

Let's define a long constant at $01FF in segment 2, and call it "SSResult." To define a long integer constant, select I integer from the Type pop-up menu in the Define Constant dialog, then enter "4" for the Size. (The shortcut key for defining a long integer at the selected

line is OPTION-L). If you look at segment 3, you'll see that the disassembler changes the code at $0038 to "sty SSResult+2," "stx SSResult." Since SSResult is a long value starting at address $02/01FF, the disassembler knows that $02/0201 is two bytes into the long, and thus generates the appropriate operand for the "sty" instruction.

Continuing on, the program creates a new menu bar, and sets it up appropriately:

```
003E: 48              pha
003F: 48              pha
0040: F40200          pea   $0002
0043: F40000          pea   $0000
0046: F40100          pea   $0001
0049: F40000          pea   $0000
004C: F40000          pea   $0000
004F: A20F43          _NewMenuBar2
0056: A20F12          _SetSysBar
005D: F40000          pea   $0000
0060: F40000          pea   $0000
0063: A20F39          _SetMenuBar
006A: F40100          pea   $0001
006D: A2051E          _FixAppleMenu
0074: 48              pha
0075: A20F13          _FixMenuBar
007C: 68              pla
007D: 8D0302          sta   |$02/0203
0080: A20F2A          _DrawMenuBar
0087: 6B              rtl
```

The "_FixMenuBar" call returns the height of the menu bar, so define an integer constant at location $0203 in segment 2, and call it "MBarHite." That's the end of segment 3.

It should now be obvious that this subroutine (and in fact the whole third segment!) simply starts the tools used by the program, so let's call it "StartUp." Use the `Define Label` command to put a START label on the first line of segment 3.

Close segment 3's window by selecting the `Close` command (ÖW) from the `File` menu, and then return to segment 2.

After the "jsl >StartUp," there is a "bcs $000B" instruction. Since the StartUp subroutine sets the carry flag to indicate an error, let's define a label at $000B called "StartErr."

The next thing the program does is to "jsr $001B," so let's look at $001B. You can either scroll the window down, or use the `Goto` command from the `Find` menu.

Put a START directive at $001B, but don't label it yet. The subroutine initializes the cursor, then calls another subroutine at $00B7. Use the `Save Location` command from the `Find` menu to save our place, then use the `Goto` command to go to address $00B7:

```
00B7: F40000            pea   $011D|-$10
00BA: F41D01            pea   $011D
00BD: 203E01            jsr   $013E
00C0: A90000            lda   #$0000
00C3: 48                pha
00C4: 48                pha
00C5: 48                pha
00C6: 48                pha
00C7: 48                pha
00C8: 48                pha
00C9: F40000            pea   $00F7|-$10
00CC: F4F700            pea   $00F7
00CF: 48                pha
00D0: 48                pha
00D1: F40200            pea   $0002
00D4: F40000            pea   $0000
00D7: F41010            pea   $1010
00DA: F40E80            pea   $800E
00DD: A20E61            _NewWindow2
00E4: FA                plx
```

Put another START directive at $00B7, but again, don't label it yet. Notice that the subroutine pushes an address then calls yet another subroutine. We can assume that it's pushing an address because an operand of the form "operand|-$10" only occurs when pushing the high word of an address (except in OBJ files, where almost any expression can occur). Let's look at $011D to see what's there. Use `Save Location`, then `Goto` $011D:

```
011D: 546869            mvn   $690000,$680000
0120: 7320              adc   ($20,s),y
0122: 6D6573            adc   $7365
0125: 7361              adc   ($61,s),y
0127: 6765              adc   [$65]
0129: 207769            jsr   $6977
012C: 6C6C20            jmp   ($206C)
012F: 626520            per   $2197
0132: 7570              adc   $70,x
0134: 7065              bvs   $019B
0136: 7220              adc   ($20)
0138: 6361              adc   $61,s
013A: 7365              adc   ($65,s),y
013C: 2100              and   ($00,x)
013E: 3B                tsc
013F: 0B                phd
0140: 5B                tcd
0141: A00000            ldy   #$0000
0144: E220              SHORT M
0146: B703              lda   [$03],y
```

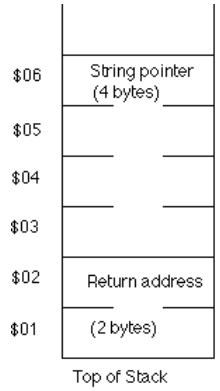
The instructions at $011D look pretty strange, causing us to assume that they probably aren't executable code, but data. To check this assumption, use the `View hex` command from the `Mode` menu. Aha! A readable string appears ("This message will be upper case!") followed by a zero byte, implying a C string. A Pascal format string would have had a length byte before the string, and no terminating zero. Click and drag to select addresses $011D through $013D. Note that in hex view, selections are made by byte, not by line as in the normal view. Now select `Define Constant` from the `Define` menu. Select `C string` from the `Type` pop-up menu, then click the `Define` button. To view the results, select `View hex` again to return to the normal disassembly mode. The strange instructions have been replaced by a C string. Let's label

the string "Message" and return back from whence we came using `Restore Location` command from the `Find` menu.

Now that we know what's being pushed, let's look at what's being called. `Save Location` again, and `Goto` the subroutine at $013E. Put a START directive at $013E:

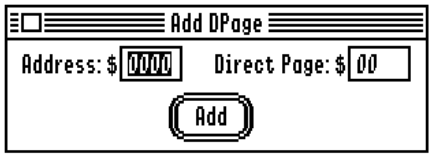
```
013E:                   start
013E: 3B                tsc
013F: 0B                phd
0140: 5B                tcd
0141: A00000            ldy   #$0000
0144: E220              SHORT M
0146: B703              lda   [$03],y
0148: F008              beq   $0152
014A: 205A01            jsr   $015A
014D: 9703              sta   [$03],y
014F: C8                iny
0150: 80F4              bra   $0146
0152: C220              LONG  M
0154: 2B                pld
0155: FA                plx
0156: 7A                ply
0157: 7A                ply
0158: DA                phx
0159: 60                rts
015A: C96190            cmp   #$9061
```

The first thing this subroutine does is to save the current direct page register, and point the direct page to the top of the stack. This is generally called creating a new stack frame. On the Apple IIGS, a subroutine often uses the direct page register in this manner to gain more flexible access to parameters passed on the stack, and/or to provide local variable space. Most Apple IIGS compilers use this technique. Let's look at the stack after executing these instructions:



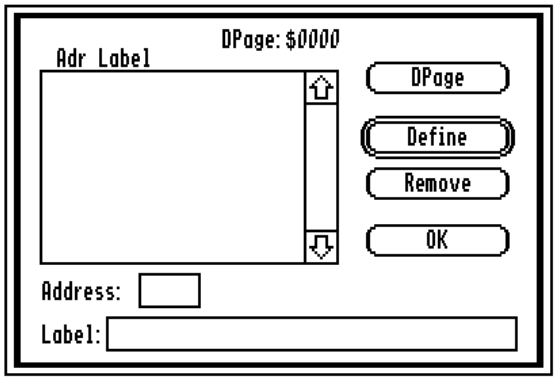
By assigning the direct page register to the stack pointer, the subroutine can use the string pointer at what is now direct page location 3 to access the string. We therefore want to give this location the name “SPtr,” for string pointer, but we want the label to be unique to this stack frame.

This is the reason that the disassembler allows for multiple direct pages, and the first thing we'll do is to inform the disassembler that this is what's happening. Select line $013E, then select `Add DPage` from the `Define` menu. It brings up a dialog like this:



When creating a new direct page, we must give it a number for reference. Notice that the direct page number (`Direct page: $`) comes up preset to 00. Since most programs have a sort of "global" direct page that's used by most of the routines, the default direct page number is zero. We could refer to this new direct page as DPage 1, but it's often more convenient to label it with the address where it is first used. Let's change the direct page number to $013E, and click `Add`. Notice that the disassembler inserts a comment before $013E that reads "; Use direct page $013E." This is to remind us that all direct page accesses from this point on will refer to labels from direct page $013E, which we just created.

Now let's add our label. Select `Edit Direct Page` from the `Define` menu. Its dialog is shown below.



At the very top of the `Edit Direct Page` dialog, we see that the current direct page being edited is area zero. There is always a DPage 0, even if we don't use it at all. We want to edit direct page number $013E, so click the `DPage` button to switch. Now, since we want to label direct page location 3, enter 3 in the `Address` box, enter "SPtr" in the `Label` box, and click `Define`. We're finished, so click the `OK` button.

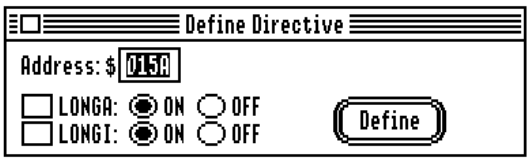
Now, notice the SHORT M and LONG M macros at lines $0144 and $0152. These are macros inserted by the disassembler just like it automatically inserts macros for toolbox calls. Select `Toolbox macros` from the `Mode` menu to see what these two macros replace. Any time the disassembler sees a sep or rep instruction that only changes the size of the accumulator or the index registers (or both), it replaces the sep or rep instruction with the appropriate LONG or

SHORT macro call (if `Toolbox macros` is turned on). Now select `Toolbox macros` again to turn the mode back on.

The core of this subroutine loops through all of the characters in a string, calling a subroutine for each character. Let's label the top of the loop “Loop” (at $0146), and the end of the loop “Done” (at $0152). Notice that the subroutine is being called with the disassembler still in short accumulator mode, since that will be important as we examine it. Save the current location, go to $015A, and put a START directive at $015A:

```
015A:                start
015A: C96190         cmp   #$9061
015D: 06C9           asl   $C9
015F: 7B             tdc
0160: B002           bcs   $0164
0162: 29DF60         and   #$60DF
0165: 48             pha
0166: F40000         pea   $0181|-$10
0169: F48101         pea   $0181
016C: A90000         lda   #$0000
016F: 48             pha
0170: 48             pha
0171: A2151A         _CautionAlert
0178: B006           bcs   $0180
017A: 68             pla
017B: 3A             dec   a
017C: 3A             dec   a
017D: D001           bne   $0180
017F: 38             sec
0180: 60             rts
```

This subroutine was called in short accumulator mode, but looking at the code makes it obvious that the disassembler thinks we're in long mode; we need to tell the disassembler to disassemble this section of code assuming an 8-bit accumulator. Select line $015A, then select the `Define Directive` (⌘!) command from the `Define` menu. Its dialog looks like this:



We want just a LONGA OFF here, so click the check box beside `LONGA`, click the `OFF` radio button to its right, then click the `Define` button. At the end of the subroutine we want LONGA back on, so enter 164 for the address, click the `ON` radio button to the right of `LONGA`, then click the `Define` button again. Close the `Define Directive` dialog.

This subroutine checks whether the accumulator contains ‘a’ through ‘z,’ and if so “shifts” it to upper case. Let's label the subroutine “ShiftA”, and label $0164 “NotLC.”

Now `Restore Location` to return to the subroutine at $013E.

The last bit of code in this routine is “pld, plx, ply, ply, phx, rts.” The pld restores the direct page to what it was before, so let's use `Add DPage` to switch back to direct page number zero. Select line $0154, then select `Add DPage` from the `Define` menu. Set the direct page to 0, then click the `Add` button.

Returning to the subroutine, add a comment after the plx that reads “pull return address,” another comment after the first ply ($0156) that reads “throw away the string address,” and a third comment after the phx ($0158) that reads “push the return address.”

Since this routine shifts a string, let's call it “ShiftStr.” Add a label at $013E called “ShitfStr.”

Now `Restore Location` to return to the previous subroutine, at $00B7.

At $00C0, the program pushes a lot of values onto the stack, then calls NewWindow2. These are all parameters for the NewWindow2 call. Let's comment them:

```
$00C3  “result”
$00C5  “title pointer (not used)”
$00C7  “refCon”
$00C9  “content draw routine”
$00CF  “window definition procedure (not used)”
$00D1  “paramTableDesc (resource)”
$00D4  “resource id”
$00DA  “resource type (rWindParam1)
```

After the NewWindow2 call, the program pulls the result into the X and Y registers, then branches to location $00F6 if there was an error. Label $00F6 as “Error.” Otherwise, it saves the window pointer at direct page locations $10 and $12, then sets the current port to the new window, and returns:

```
00DD: A20E61          _NewWindow2
00E4: FA              plx
00E5: 7A              ply
00E6: B00E            bcs   $00F6
00E8: 98              tya
00E9: 8512            sta   $12
00EB: 8610            stx   $10
00ED: 48              pha
00EE: DA              phx
00EF: A2041B          _SetPort
00F6: 60              rts
00F7: A91400          lda   #$0014
00FA: 48              pha
00FB: 48              pha
00FC: A2043A          _MoveTo
0103: F40000          pea   Message|-$10
0106: F41D01          pea   Message
0109: A204A6          _DrawCString
0110: 6B              rtl
0111: D412            pei   $12
```

Let's label the direct page locations used. Select `Edit direct page` from the `Define` menu. Since this is the program's default direct page, we'll use direct page zero. We'll call location $10 WPtr, so enter 10 for `Address`, “WPtr” for `Label`, and then click the `Define` button. Now, since the disassembler doesn't know how large the direct page values are, it doesn't know that WPtr is a long, and that location $12 should be “WPTR+2,” so we have to tell it. Enter 12 for `Address`, “WPtr+2” for `Label`, then click the `Define` button again. Click the `OK` button when finished.

Let's call this routine “CreateW,” so put that label at $00B7.

Now let's look at the routine defined as the window's content draw routine, at $00F7. `Goto` $00F7, and give it a START label of "DrawW." This routine simply prints the string at Message in the window:

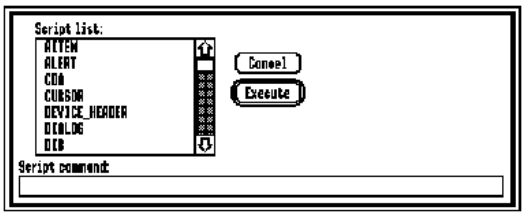
```
00F7:            DrawW    start
00F7: A91400              lda   #$0014
00FA: 48                  pha
00FB: 48                  pha
00FC: A2043A              _MoveTo
0103: F40000              pea   Message|-$10
0106: F41D01              pea   Message
0109: A204A6              _DrawCString
0110: 6B                  rtl
0111: D412                pei   WPtr+2
0113: D410                pei   WPtr
0115: A20E0B              _CloseWindow
011C: 60                  rts
011D: 5468697320 Message  dc    c'This message will be upper case!'
013D: 00                  dc    h'00'
                          end
013E:            ShitfStr start
013E:            ; Use direct page $013E
013E: 3B                  tsc
013F: 0B                  phd
```

We notice, however, that Message is accessed from two different routines (CreateW and DrawW), so let's put it in a DATA area. Select $011D, then select `Define Label` from the `Define` menu. Enter “WData” for the label, click the `DATA` radio button, then click the `Define` button.

Now use `Restore Location` to return to the subroutine at $001B. After calling CreateW, it returns if the carry flag is set, so let's label $0028 “GotW.”

Now the program zero's a location, then branches if it's not zero. Since we don't yet know the purpose of that location, we'll continue on.

Next the program pushes some parameters, and then calls TaskMaster. The last parameter pushed is a pointer to a task record. `Save Location`, and `Goto` $0207. Let's use a script to define the task record. Select the line at $0207, then select `Execute Script` from the `Scripts` menu. Its window looks like this:



Find `TaskRecX` in the scrolling list and select it, then click the `Execute` button. The TaskRecX script is used to define an extended task record.

To finish off, we'll label the following locations (the labels are adapted from the *ToolBox Reference* manual):

```
$0207  “taskRec”
$0209  “wmMsg”
$020D  “wmWhen”
$0211  “wmWhere”
$0215  “wmMods”
$0217  “wmTData”
$021B  “wmTMask”
$021F  “wmLastClick”
$0223  “wmClicks”
$0225  “wmTData2”
$0229  “wmTData3”
$022D  “wmTData4”
$0231  “wmLastWhere”
```

Two of the labels are too long to fit in the label field. You can enter them by double-clicking in the label field and typing, but you won't be able to see all of the characters until you click on another line. If you prefer to see what you are typing, you can always use the Define Label dialog, which allows you to see more characters.

Use `Restore Location` to return to the subroutine.

The return value from TaskMaster is a two-byte integer, called the taskCode:

```
003A: A20E1D          _TaskMaster
0041: 68              pla
0042: F0E7            beq   $002B
0044: C92200          cmp   #$0022
0047: B0E2            bcs   $002B
0049: 0A              asl   a
004A: AA              tax
004B: FC5800          jsr   ($0058,x)
004E: 80DB            bra   $002B
0050: 206501          jsr   $0165
0053: B0D3            bcs   GotW
0055: 4C1101          jmp   $0111
0058: 9C009C          stz   $9C00
005B: 009C            brk   $009C
005D: 009C            brk   $009C
005F: 009C            brk   $009C
0061: 009C            brk   $009C
0063: 009C            brk   $009C
0065: 009C            brk   $009C
0067: 009C            brk   $009C
```

The program range-checks the value of taskCode, and loops back to $002B if it's $00 or greater than $22, so label location $002B “Loop.” The program then uses a jump table to call handlers for each taskCode. Let's label $0058 “JumpTbl.” Now we need $22 two-byte integer constants for the table. Select line $0058, then issue the `Define Constant` command. $58 + $21 * 2 = $9A, so enter 9A for `End Address`. Make sure the `Type` is `I integer` and that its `Size` is 2, then click the `Define` button:

```
0055: 4C1101              jmp   $0111
0058: 9C009C009C JumpTbl  dc    i2'$009C,$009C,$009C,$009C'
0060: 9C009C009C          dc    i2'$009C,$009C,$009C,$009C'
0068: 9C009C009C          dc    i2'$009C,$009C,$009C,$009C'
0070: 9C009C009C          dc    i2'$009C,$009C,$009C,$009C'
0078: 9C009D009C          dc    i2'$009C,$009D,$009C,$009C'
0080: 9C009C009C          dc    i2'$009C,$009C,$009C,$009C'
0088: 9C009D009C          dc    i2'$009C,$009D,$009C,$009C'
0090: 9C009C009C          dc    i2'$009C,$009C,$009C,$009C'
0098: 9C009C              dc    i2'$009C,$009C'
009C: 60                  rts
009D: AD1702              lda   wmTData
00A0: C91502              cmp   #$0215
00A3: D003                bne   $00A8
00A5: CE0502              dec   $0205
00A8: F40000              pea   $0000
00AB: AD1902              lda   wmTData+2
00AE: 48                  pha
00AF: A20F2C              _HiliteMenu
00B6: 60                  rts
```

Most of the table entries point to location $009C, which is simply an rts instruction. Put a START directive at $009C, and label it “Ignore.” Only two entries – corresponding to taskCodes $11 and $19 – do not point to Ignore. TaskCode $11 is wInMenuBar, and $19 is wInSpecial. Both are returned by TaskMaster when the user makes a menu selection, so let's call $009D “DoMenus” and put a START directive there.

WmTData holds the menu item selected; the routine checks whether item $0215 was selected. By inspecting the resource fork of Sample (using Derez®, or a similar tool), we would see that item $0215 is Quit. If Quit was selected, the program decrements a value at location $0205. The value appears to be a flag, so define a two-byte integer constant at $0205 called “QFlag.” Now label $00A8 “NotQuit.” The last thing this routine does is to unhighlight the menu title (wmTData+2 contains the menu number selected).

Let's go back up to $001B. Now we see that the program was checking whether Quit was selected at $002B, so label $0050 “Done.” If the user selects Quit, the program calls $0165, so `Save Location`, `Goto` $0165, and put a START directive there:

```
0165:                     start
0165: 48                  pha
0166: F40000              pea   $0181|-$10
0169: F48101              pea   $0181
016C: A90000              lda   #$0000
016F: 48                  pha
0170: 48                  pha
0171: A2151A              _CautionAlert
0178: B006                bcs   $0180
017A: 68                  pla
017B: 3A                  dec   a
017C: 3A                  dec   a
017D: D001                bne   $0180
017F: 38                  sec
0180: 60                  rts
0181: 1E00AA              asl   $AA00,x
0184: 0054                brk   $54
0186: 00D6                brk   $D6
0188: 0161                ora   ($61,x)
018A: 1E8080              asl   $8080,x
```

This subroutine calls CautionAlert with an alert template at $0181. Select $0181, then select `Execute Script` from the `Scripts` menu. Select `Alert` from the list, then click the

`Execute` button. Looking over the template created at $0181, we notice that the script did not define the strings at $01B7, $01D2, or $01F1, so define P strings at each of those locations.

We'll let the disassembler generate labels for all of the labels in the alert template, but we will label $0181 “ATempl.” We'll also label $0180 “Retn,” and $0165 “ChkQuit.”

`Restore Location`. If ChkQuit returns with carry clear, the routine jumps to $0111. `Save Location`, `Goto` $0111, and place a START directive there:

```
0111:                       start
0111: D412                  pei   WPtr+2
0113: D410                  pei   WPtr
0115: A20E0B                _CloseWindow
011C: 60                    rts
                            end
011D:             WData     data
011D: 5468697320  Message   dc    c'This message will be upper cas
013D: 00                    dc    h'00'
                            end
013E:             ShitfStr  start
013E:             ; Use direct page $013E
013E: 3B                    tsc
013F: 0B                    phd
0140: 5B                    tcd
0141: A00000                ldy   #$0000
0144: E220                  SHORT M
0146: B703        Loop      lda   [SPtr],y
0148: F008                  beq   Done
014A: 205A01                jsr   ShiftA
```

This routine is short and easy. It simply pushes the pointer to the window, and calls CloseWindow. Let's call the routine “CloseW.”

`Restore Location` back to $001B, and call this subroutine “Main,” since it does the main work of the program.

Go all the way back to $0000. Looking at the beginning of the program, we see that after calling Main, the program calls $0235. `Save Location`, `Goto` $0235, and place a START directive there:

```
0235:                   start
0235: AD0102            lda   SSResult+2
0238: AEFF01            ldx   SSResult
023B: D003              bne   $0240
023D: A8                tay
023E: F00C              beq   $024C
0240: F40100            pea   $0001
0243: 48                pha
0244: DA                phx
0245: A20119            _ShutDownTools
024C: ADFD01            lda   userID
024F: 48                pha
0250: A20203            _MMShutDown
0257: A20103            _TLShutDown
025E: 60                rts
                        end
```

This routine skips a call to ShutDownTools if SSResult is zero, so label $0240 “Started,” and $024C “Not.”

This routine obviously just shuts down the tools, so call $0235 “ShutDown,” then `Restore Location`.

The last thing the program does is quit by making a GS/OS quit call. The parameter list for the quit call is at $0019. Select $0019 then select `Execute Script` from the `Scripts` menu. Choose `GSOS_Quit` from the scripts list, then click the `Execute` button. Label $0019 “QParms.”

Now we're almost done. Choose `Generate Labels` from the `Define` menu. This causes the disassembler to generate labels for any locations that weren't assigned labels. It's a good idea to use this command before creating a source file, even if you think you put in all the labels yourself. You may well have missed some.

In looking over the program, we might notice that we need a DATA area at $01FB, so define one there, and call it “Globals.”

Only two more steps remain. First select `Save Template` from the `File` menu, and save this template as “sample.t” in case you need to change something later. In point of fact, it is a good idea to periodically save a template as you disassemble, to avoid having to redo work from scratch in case you get boggled down in deep code, and make some erroneous assumptions about what the program is doing.

To create source for this program, select `Create Source File` from the `File` menu. Save the source for this segment as “sample.asm.” Now select SEGMENT3 from the `Segments` menu, and select `Create Source File` again, calling the source for this segment “sample2.asm.”

Four things need to be done before the source code can be assembled: use an APPEND directive to append the source for sample2.asm to the end of sample.asm; add the line “mcopy sample.macros” at the beginning of sample.asm; since the disassembler doesn't automatically generate USING statements for DATA areas in other segments, add the line “using Globals” to sample2.asm, directly beneath the line containing the StartUp label; and remove the GEQUs from the beginning of sample2.asm, since these are duplicated in sample.asm.

Chapter 3
Menu Commands

In the descriptions that follow, most commands that affect the disassembly apply to the current segment. Segments, as used in this manual, refer to load segments. The current segment is the segment window in which the last command issued was performed. In the event that a dialog box is front-most on the desktop, so that none of the segment windows is highlighted, you can determine the current segment by issuing the `Segment info` command, available under the `Segments` menu. You can select a segment to be the current segment by selecting its name in the `Segments` menu. Two commands affect all of the segments in the current disassembly – `Save template` and `Lowercase opcodes`.

The File Menu

The `File` menu is used to open files, save templates to disk, create source files, print, and exit the program.

New (ÅN)

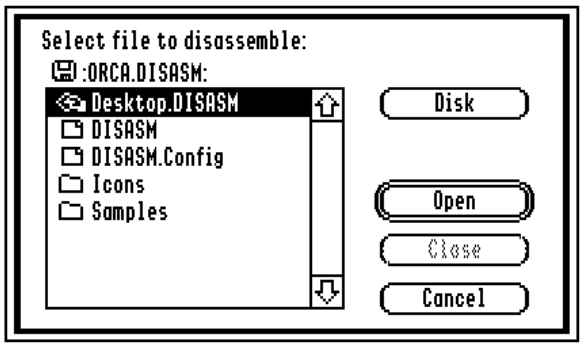
The `New` command removes the file being disassembled from memory and discards the current template.

Open... (ÅO)

The `Open` command is used to load a file to disassemble from disk. You may disassemble any standard object module format (OMF) file (i.e. files with filetypes such as S16, EXE, RTL, NDA, and CDA, except libraries), plus binary (BIN), and ProDOS 8 system (SYS) files.

When you open a file, all of its segments will be read into separate windows, and the first segment window will be opened on the screen. The `Segments` menu will contain a list of all of the segments in the file. If a file is expressed (contains an expressload segment: "~ExpressLoad" or "EXPRESSLOAD"), that segment cannot be viewed since it contains no source code. The menu item `ExpressLoad` will appear in the `Segments` menu, but it will be dimmed and not selectable.

The `Open...` command causes a standard Apple IIGS Open dialog box to appear on the screen. The `Open` dialog looks similar to the one on the next page.



At the top of the dialog box is a message explaining the function of the dialog (`Select file to disassemble`).

Beneath the message is an icon and the name of the object the icon depicts. In the dialog above, this is a disk named ORCA.DISASM. If the current prefix is within a folder, you can click on the icon's name to move up one directory level.

Below the icon is a list of files that can be opened for disassembly. The list contains only those files having a filetype supported for disassembly; thus, the list doesn't necessarily show all of the files that may reside in the current prefix. A file or folder is selected for opening by clicking the mouse on its name.

Next to the list of files are four buttons. The top button, labeled `Disk`, is used to move through the disks currently available on your computer. Clicking on the `Disk` button with the mouse causes the current prefix to be set to the next disk on the system.

The second button is labeled `Open`, and is outlined. The outlining indicates that this is the default button for the dialog – you can click on the button with the mouse or simply press the RETURN key to open a file or folder.

The third button is labeled `Close`. Clicking on it causes the currently open folder to be closed, and the prefix to be set to the folder above the current folder. The `Close` button will be dimmed when the current prefix is not within a folder.

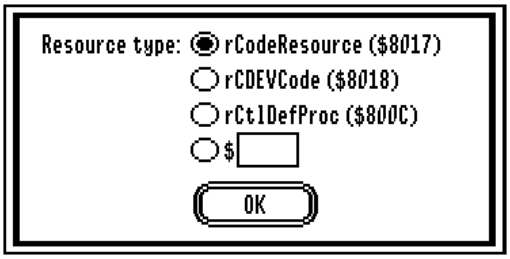
The last button is labeled `Cancel`. Clicking on this button causes the dialog box to go away, with no action being taken by the disassembler. Note: If you have used the `Disk` and/or `Open` buttons to change the current prefix, the current prefix will *not* be reset to the value it had before issuing the `Open` command.

Open code resources...

The `Open code resources` command is used to load a file containing a resource fork from the disk. The disassembler supports code resources only, including "generic" code, type $8017; CDEV code, type $8018 (a CDEV is a control panel device – these are loaded by the control panel NDA from the CDEVS directory of the SYSTEM folder); control definition procedure code, type $800C; and other standard or user-defined types.

When `Open code resources` is first invoked, it brings up a standard open file dialog box, similar to that shown above for the `Open` command. Only files containing a resource fork

will be displayed in the file list. If a file with a resource fork is selected for opening, the command then brings up this dialog to ask the type of resource you want to disassemble:



All code resources of the type specified will be loaded, with each resource treated as though it were a unique segment of a load file (i.e. each resource will be loaded into its own window, and available for selection from the `Segments` menu).

To "disassemble" data resources, use DeRez® or some similar tool.

Close (ÓW)

The `Close` command closes the front window. The front window is the window that is currently highlighted. Dialog boxes are considered windows.

Closing a segment window simply hides it from view, and does not alter its contents in any way. When you want to work with a closed segment window, simply choose that segment from the `Segments` menu.

Dialog boxes that are closed retain their position and will appear again in the same place when reopened.

Load Template... (ÓT)

The `Load Template..` command loads a previously saved template from disk. The template includes all labels, direct page labels, constants, directives, comments, etc., that have been specified for a file. The `Load Template...` command brings up a standard `Open` dialog box, similar to that described in the `Open...` command above. As with the `Open...` command, the list of files displayed that can be loaded are restricted. Template files generated prior to version 1.2 of ORCA/Disassembler have a filetype of BIN with an auxtype of 0. Starting with version 1.2, template files have a filetype of $5E (development utility) and an auxtype of $8001. Only these two types of files will be shown in the file list in the `Load Template` dialog.

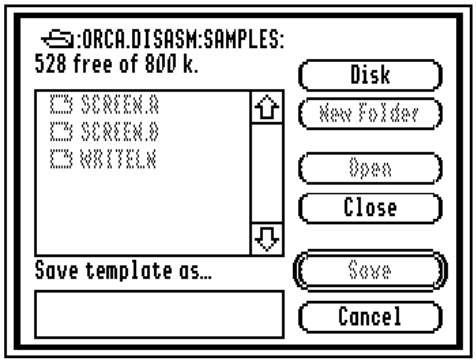
Save Template (ÓS)

The `Save Template` command saves the current file's template on the disk. If the template was loaded from disk, or has already been saved at least once, then ORCA/Disassembler knows the name and location of the template file on disk. If you issue the `Save Template`

command for a new template, it will function as though you had selected the `Save Template As...` command.

Save Template As...

The `Save Template As...` command is used to write the current template to a file that is different from the original template file, or to save a new template for the first time. The `Save Template As...` command brings up a standard `Save` dialog similar to that pictured below.



At the top of the dialog box is an icon indicating the current prefix, followed by the pathname of the current prefix. If the current prefix is within a folder, you can click on the pathname to move up one directory level.

Beneath the current prefix is a message giving the amount of free space on the disk, as well as the total amount of disk space.

Below the free space message is a box containing a list of files and folders in the current prefix. Unlike the `Open` and `Load Template` file lists, this list shows all of the files and subdirectories in the current prefix. Note that filenames are dimmed and not selectable, whereas folders can be selected. The first folder in the list (if there are any folders in the list) will be highlighted as the default folder to select.

Beneath the file list is the message `Save template as...`, followed by a box containing a flashing vertical bar. The box is where you type in the name of the template file. The flashing vertical bar is called the insertion point – it is the cursor for the box, called a line-edit box.

Next to the file list are six buttons. The `Disk` button is used to move through the disks currently on-line.

The `New Folder` button is used to create a new folder within the current prefix. To create a new folder, use the `Disk` and `Open` buttons to move to the prefix where you want to create the new folder. Now enter the name of the folder in the `Save template as...` box. Finally, click on the `New Folder` button to create the folder. At this point you can use the `Open` button to open this folder if the template file is to be placed within this folder.

The `Open` button is used to open a folder. The button will be dimmed and unselectable if there is no folder to open.

The `Close` button is used to close an open folder. The button will be dimmed and unselectable if the current prefix is the root volume of a disk.

The `Save` button is used to save the template file to disk. It is outlined since it is the default button for this dialog.

The `Cancel` button is used to cancel the `Save Template As...` command, without creating a new template file on disk. Note that the prefix will remain set to whatever it was at the time the `Cancel` button is clicked. That is, if you have used the `Disk` and/or `Open` buttons to change the current prefix, then the current prefix will *not* be reset to the value it had before issuing the `Save Template As...` command. Also, any folders you have created prior to clicking on the `Cancel` button will remain on the disk.

Create Source File...

The `Create Source File...` command generates a source file for the current segment. Like the `Save Template As...` command, this command brings up a standard Apple IIGS `Save` dialog for you to specify the name for the new source file.

The `Create Source File...` command will generate up to approximately 48K bytes of source code in the file specified, and will automatically create additional source files by appending a ".A," ".B," etc., to subsequent source filenames. Filenames used by the ProDOS FST are limited to 15 characters. If the appended filename suffix (the ".A," ".B," ...) causes the filename to exceed 15 characters, the disassembler will truncate the original filename by dropping the last two characters so that the original filename, minus the last two characters, plus the two-character suffix, totals to exactly 15 characters.

ORCA/Disassembler attempts to break long source files on subroutine boundaries, and automatically inserts ASM65816 APPEND directives at the end of each source file. If the source code has been broken up into separate files, and the disassembler has had to truncate the names of the source files in order to fit the appended suffixes into 15 characters, the filenames in the APPEND directives will *not* reflect the truncation. You will need to edit each of the source files' APPEND directives before assembling your source code.

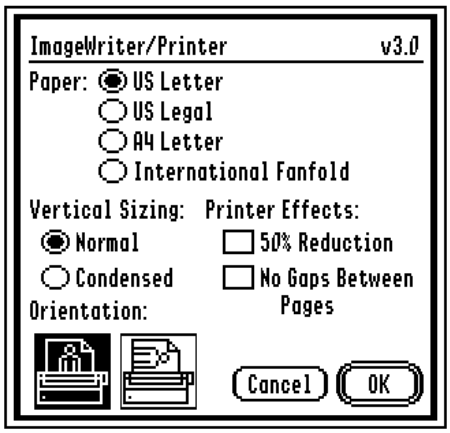
ORCA/Disassembler automatically generates USING directives for code segments that reference named objects defined within DATA segments during creation of the source file. It also places GEQU directives into the source file for the named constants it uses in the disassembly.

The source file the disassembler creates will usually only require being run through the ORCA/APW MACGEN utility to generate a macro file before being ready to assemble.

You can cancel the `Create source file...` command with open-Apple period (**.**) – hold down the key, then press the **.** key.

Page Setup...

The `Page Setup...` command brings up a standard Apple IIGS `Page Setup` dialog box, similar to the one shown here (the actual dialog depends on the printer you have chosen from the Control Panel NDA):

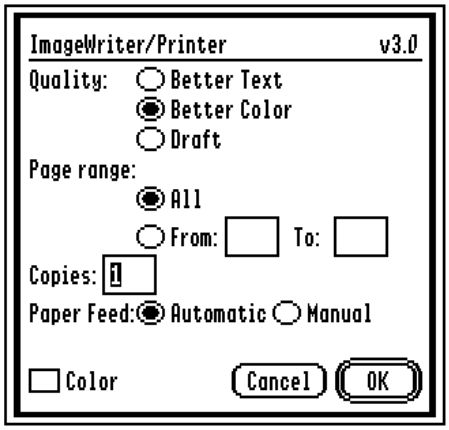


There are four radio buttons at the top of the dialog which allow you to tell the system the type of paper you will use to print your document. The `Vertical Sizing` buttons let you specify whether the typeface is large (`Normal`) or small (`Condensed`). The `Printer Effects` check boxes allow you to shrink the overall size of each printed page by one half and/or have the end of each page be followed immediately by the next page without extra blank lines. The vertical or horizontal orientation you prefer is selected by clicking on the appropriate icon.

Print... (P)

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

The `Print` command brings up a standard dialog similar to the one below.



There are radio buttons at the top of the dialog which let you choose the quality of the printing. Although you could select one of the `Better` buttons, we recommend `Draft` for faster printing.

The `Page range` parameter is ignored by ORCA/Disassembler.

The `Copies` box lets you specify the number of copies to print, while the `Paper Feed` option allows you to choose between automatic or manual feed.

The `Color` check box lets you inform the system whether or not you have a color printer.

Clicking the `OK` button causes printing to start, while selecting `Cancel` just cancels the `Print...` command.

You can stop the printing by pressing open-Apple period (.) – hold down the key and then press the `.` key.

Quit (Q)

All windows on the desktop are closed. If you've changed the current template since the last time it was saved, you are presented with an alert box that gives you the chance to save the template information or cancel the `Quit` command. The alert box is depicted below.

If you click on the `DISCARD` button, the template information is not saved.

If you choose to save the template information to disk, you are presented with the `Save Template As...` dialog box if the template file is new, or else the new template information replaces the old template file if the template information had been previously saved to disk.

Selecting the `Cancel` button cancels the `Quit` command.

The Edit Menu

The `Edit` menu provides the standard editing capabilities common to virtually all desktop programs. Although ORCA/Disassembler does not make extensive use of the `Edit` menu items, they can be used by New Desk Accessories and dialogs with line-edit items.

Undo

This command is not currently supported by ORCA/Disassembler, but may be used by New Desk Accessories.

Cut, Copy, Paste, Clear

These four standard editing commands can be used with NDAs and when you are editing text in a line-edit box within a dialog box.

Select All (Ó A)

Selects the entire current segment. If the `Edit Script File` window is active, `Select All` will select all of the text in the window.

The Define Menu

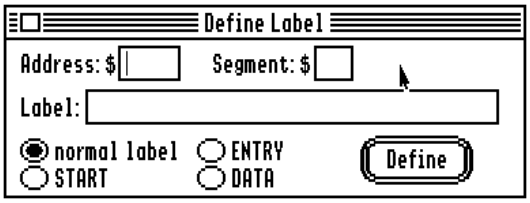
The `Define` menu is used to define labels, directives, constants, and comments; remove all of the above; edit the direct page variable list; add relocation records; or have ORCA/Disassembler automatically generate labels for the file.

Most of the `Define` commands bring up dialog boxes. Except in the case of the `Edit Direct Page...` command, the dialog boxes are modeless. That is, they remain on the desktop, even after an action has been applied to the disassembly, until you specifically close the dialog box or quit the program. You can close a modeless dialog box by clicking on its close box, located at the top left-hand corner of the box, or by selecting the `Close` command from the `File` menu when the dialog box is the active (highlighted) window.

Many of the dialogs ask for addresses. You should enter a hexadecimal number which specifies the offset from the beginning of the segment to the location you wish to define. Offsets are given on the far left-hand side of the disassembly as four-digit hexadecimal values. The disassembler will not allow you to load a segment larger than 64 KB.

Define Label... (Ó D)

The `Define Label` command attaches a label to an address. It brings up the `Define Label` dialog:



Beneath the title bar of the dialog is a line-edit box marked `Address`. You should enter the hexadecimal value of the offset from the beginning of the segment to the location of the label. (See the tutorial in Chapter 2 for an example showing how to find the address for a label.) If a line was selected prior to issuing the `Define Label...` command, the address box will be filled in with the address of the selected line.

Next to the address is a line-edit box marked `Segment`. Enter the hexadecimal value of the segment number which is to contain the label. You can find the segment number of the current segment by issuing the `Segment info` command, available under the `Segments` menu. The default value for the `Segment` box is the number of the current segment.

In the center of the dialog is a large line-edit box marked `Label`. Enter the name you wish to give the label. The length of the label is restricted to 19 characters.

There are four radio buttons at the bottom of the dialog which allow you to specify how the label is to be used. `Normal label` means that the label will refer to a constant or a storage area. The other buttons apply to START, DATA, and ENTRY directives.

Clicking on the `Define` button or pressing the RETURN key causes the label definition to be applied to the disassembly.

Quick Label Definition

There is a faster method of defining labels in ORCA/Disassembler than going through the `Define Label` dialog box. Simply double-click the mouse on the location in the disassembly window where the definition is to appear, in the label field, and then type the new label.

START directives can be placed before a line by first selecting the line, then typing OPTION-S (hold down the OPTION key and simultaneously press the S key). Remove the directive by again typing OPTION-S.

ENTRY directives can be placed before a line by first selecting the line, then typing OPTION-E. Remove the directive by again typing OPTION-E.

Edit Direct Page... (E)

ORCA/Disassembler allows you to assign labels to direct page locations. It assumes that the program uses one direct page area, defines it to start at $0000, and assigns it a direct page reference number of $0000. You can use the `Add DPage` command, described later in this section, to define multiple direct page areas.

In assembly language, one typically uses equates to give labels to direct page locations, as in:

```
count   equ  4
        . . .
        lda  number
        sta  count
        . . .
number  ds   2
```

If you were to disassemble this program, it would look something like this:

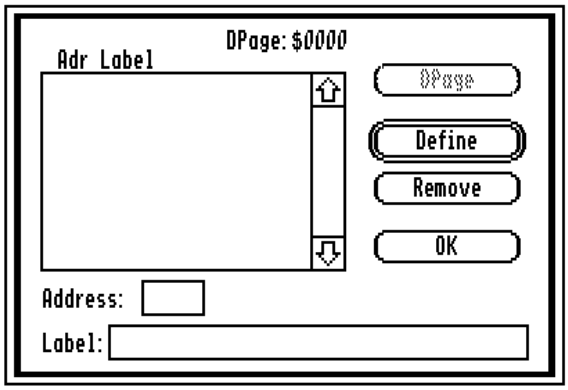
```
        lda  L0038
        sta  $04
```

where L0038 was created by the Generate labels... command. You could use the Edit direct page... command to assign a label to direct page location $04. After using this command to assigning $04 the label count, the disassembly would look like:

```
lda  L0038
sta  count
```

When you create source code for the program, appropriate equ directives are generated.

The Edit Direct Page... command brings up the Edit Direct Page dialog:



At the top of the dialog box is the message DPage:, followed by the reference number assigned to this direct page area when it was created with the Add DPage... command.

Beneath the reference number is a list of the direct page locations you have already created for this direct page area.

Below the label list is a line-edit box marked Address. Enter the one-byte hexadecimal address of the direct page location you wish to add to the list.

Beneath the Address box is a line-edit box marked Label. Enter the label name that you wish to attach to the direct page location.

Next to the label list are a series of buttons. The top button, DPage, is used to move to the direct page area you wish to edit. If there is only one direct page area, clicking on the button will have no effect.

The next button is marked Define. Clicking on this button adds a new label to this direct page area. To rename a label, click on the label in the label list, edit its name in the Label line-edit box, and then click on the Define button.

The third button is labeled Remove. To delete a label from the direct page area, click on the label in the label list, and then click the Remove button.

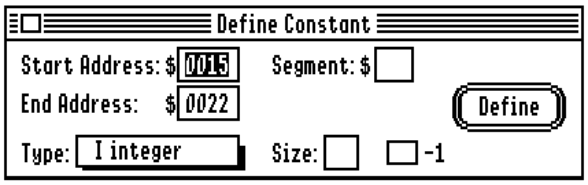
The OK button is used to exit the dialog box.

After you have entered in the direct page labels and returned to the disassembly, you will see the direct page locations changed from hexadecimal numbers to the labels you defined.

Define Constant... (K)

You use the `Define Constant...` command to tell the disassembler to treat certain bytes as data instead of code. The data can be any of the DC directive types (integer, hexadecimal, floating point, address, character, or binary), a DS directive, a Pascal-type string (leading length byte followed by sequence of characters), a C-style string (sequence of characters followed by a zero byte), or a C1 string (a GS/OS class 1 string, a two-byte integer size followed by a sequence of characters).

The `Define Constant...` command brings up a dialog, depicted below:



At the top of the dialog is a line-edit box labeled `Start Address`. Enter the offset, as a hexadecimal number, from the start of the segment to the beginning of the constant. If you have selected some text before issuing the `Define Constant...` command, the disassembler will fill in this box with the address corresponding to the start of the selected text.

Below the `Start Address` box is another line-edit box marked `End Address`. The ending address of the constant is its offset from the start of the segment. The value entered should be a hexadecimal number. If you do not enter a value, the disassembler defines it internally based on the type of constant data you select to start at `Start Address`. If you have selected some text before issuing the `Define Constant...` command, the disassembler will automatically fill in this box with the address corresponding to the end of the selected text. In the assembly-language disassembler display, the `End Address` will contain the address of the beginning of the last line selected. In the hex-mode display, `End Address` will contain the address of the last byte selected.

Next to the address boxes is a line-edit box marked `Segment`. Enter the hexadecimal number corresponding to the segment for which the constant is to be defined. You can leave this box blank if the constant is to appear in the current segment.

At the bottom of the dialog is a pop-up menu which allows you to specify the type of constant being defined. The integer, hex, float, address, character, double, binary, and extended items all refer to DC directives.

The `Pascal string` item will replace the contents of the disassembly with the ORCA/M DW macro, if the starting address you've specified truly defines the beginning of a Pascal-style string – that is, if the disassembler finds a one-byte integer length value, followed by that many ASCII characters. If the starting location you give does not correspond to the beginning of a Pascal string, the disassembler tries to resolve the one-byte character at the address to an integer, and then converts the data it finds to DC C and DC H directives. The length of the conversion is determined by the value the disassembler finds at the specified address. The converted block will be truncated if the length exceeds the length of the segment.

The `C string` item will replace the contents of the disassembly with a series of DC directives. The conversion begins at the address specified, and continues until either a zero byte or the end of the segment is encountered.

The `C1 string` menu item will replace the contents of the disassembly with a two-byte integer, followed by a series of DC C and DC H directives. The length of the conversion is determined by the value the disassembler finds at the starting address. The converted block will be truncated if the length exceeds the length of the segment.

Next to the `Type` pop-up menu is a line-edit box marked `Size`. Enter into this box the size, in bytes, of the integer (DC I or DC A) constant you wish to define. The value should be a decimal number, ranging from 1 to 8. An example of the use of the `Size` box would be defining an eight-byte integer (DC I8). The value placed into the box would be 8. The `Size` box will only be selectable when the constant you are defining is integral (DC I or DC A). If you do not enter a value into `Size`, the disassembler assumes a one-word (2-byte) integer.

The `-1` check box is used to automatically generate DC A'Label-1' entries in the disassembly. A common programming practice is to build jump tables. An address in the jump table might be used as an RTS instruction. In processing an RTS instruction, the 65816 pulls the return address from the stack, increments it, and then jumps to the incremented address. In the jump table, you would normally code the address minus one, so that when the processor increments the address, the final address will be correct.

Before applying the `-1` option in the `Define constant` dialog, you will probably see in the jump table in the disassembly a reference to the address before the label you would like to reference. After applying the `-1` option, you will see the correct Label-1 reference, assuming that you had previously defined the label in the disassembly.

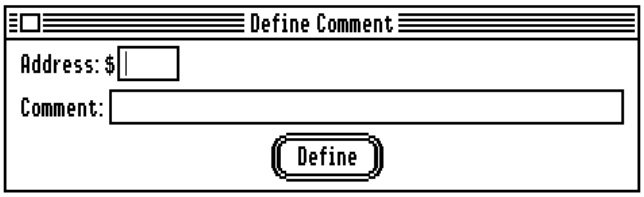
Quick Constant Definition

There are some short-cuts for defining constants in ORCA/Disassembler, allowing you to avoid the `Define Constant` dialog box. For each, you should start by selecting the line where the constant is to appear. Next, hold down the option key and simultaneously press one of the keys below, depending on the constant you wish to define:

B	define hex byte
C	define C string
G	define a GS/OS class 1 string
L	define a long (four-byte) integer
P	define a P string
R	define a rectangle (A rectangle is a record that consists of four two-byte integers.)
W	define a word (two-byte integer)

Define Comment... (;)

The `Define Comment...` command allows you to place comments into the disassembly. It brings up a dialog box pictured below:



In the `Address` box, enter the offset, as a hexadecimal number, from the beginning of the segment to the line which is to contain the comment.

In the line-edit box beneath the `Address` box, enter the comment. Comments are limited to 40 characters. The comment in the source file, unless changed with the `Tab stops...` command, will begin in column 41.

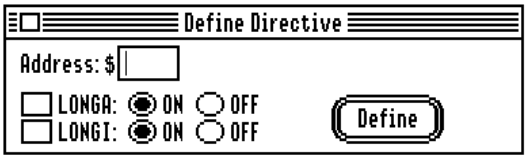
If a line has been selected when you issue the Define Comment... command, the `Address` and `Comment` boxes will default to the address and comment (if any) at the selection.

Quick Comment Definition

There is a faster way to define a comment in ORCA/Disassembler without going through the `Define Comment` dialog. Simply double-click the mouse on the location where you want to define the comment (the line can already contain a comment), and then type the new comment directly into the disassembly.

Define Directive... (!)

This command allows you to insert two of the available ORCA/APW assembler directives into your disassembly, LONGA and LONGI. It brings up a dialog box, depicted below:

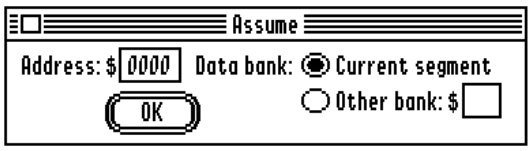


In the `Address` line-edit box, enter the offset, as a hexadecimal number, from the beginning of the segment to the location where the directive is to be inserted. If text is selected, the default value for the `Address` box will be the starting address of the selection.

Below the `Address` box is a list of directives that can be inserted. Select the directive you wish to insert by clicking on the check box of the desired directive. Next click the ON or OFF radio button, as appropriate.

Assume...

The `Assume...` command is used with non-OMF files only. It causes the disassembler to "assume" a particular data bank for all short (absolute) operands, starting at the specified address. Its dialog looks like this:



Selecting the `Current segment` radio button causes the disassembler to "assume" that the data bank register is set to the current code bank. Selecting the `Other bank` radio button allows you to enter the number of the data bank, as a hexadecimal value, in the line-edit box next to `Other bank`.

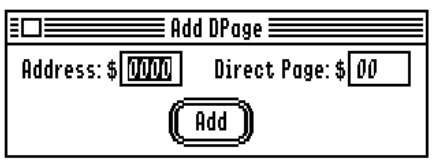
The data bank information is not saved in the templates for OMF files, so do not use ASSUME with OMF files.

Add DPage...

The `Add DPage...` command is used to create a new direct page area for the disassembly. You would typically create multiple direct page areas when disassembling a program that sets up a new direct page area upon entry to every subroutine. Most high-level languages will generate code that uses the stack for local variables. On the 65816, it is easiest to address the values on the stack by setting up a direct page area based on the value of the stack pointer upon entry to the subroutine. The Direct Page register is then restored before exiting the subroutine. This sequence of instructions is typical:

```
PHD           save old Direct Page register
TSC
CLC
SBC  #10      get 10 bytes of storage from the stack
TCS
TCD           set up new direct page from stack
```

You can use the `Add DPage...` command to set up new direct page areas throughout your disassembly. The `Edit direct page...` command, described earlier in this section, allows you to assign labels to direct page locations. `Add DPage` brings up the dialog box:



In the Address box, enter the offset, as a hexadecimal value, from beginning of the segment to the location where the direct page area is to be used. If text is selected, the default value for this box will be the beginning address of the selection.

In the Direct Page box, enter a reference number, as a hexadecimal value, of the direct page area. We recommend that you use $0000 for the "normal" direct page, since it is the default, and then use the address at which you are defining the new direct page for the reference number for alternate direct pages.

After you have created a second direct page area for the current segment, the disassembler will place a comment, similar to the one depicted below, into the disassembly at the location where the direct page is to be applied:

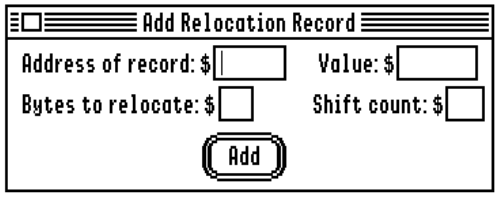
```
0006: A91500              lda   #$0015
0009:             ; Use direct page $0009
0009: 48                  pha
000A: A20C20
```

where DPAGE is a dummy directive and its operand 000A is the reference number you assigned to the direct page area.

See the Edit direct page... command, described earlier in this section, for more information about ORCA/Disassembler's use of direct page areas.

Add Relocation Record...

This command is used to add a new relocation entry for a specified address. It is generally only used when disassembling non-OMF files, since standard OMF files contain relocation information. It brings up the dialog box:



The Address of record box is used to specify the address of a reference to a label that needs to be relocated.

The Value box specifies the address of the label, at the location in the segment where it is defined.

The Bytes to relocate box should contain the size of the machine code where the label is defined.

The Shift count box is a one-byte value giving the amount that the relocated address needs to be shifted. A positive value indicates shifting to the left; a negative value indicates shifting to the right.

As an example, suppose the segment contained these two lines:

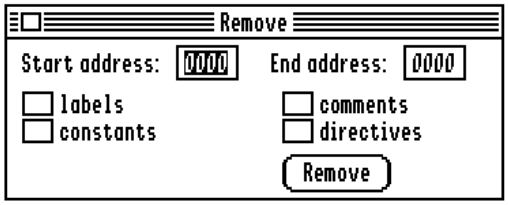
```
address       source code

$0042         MyLabel       DS     8
                            . . .
$1000                       LDA    MyLabel|2
```

The `Address` box should be filled in with $1001, since the LDA instruction is one byte long. The `Value` box should contain $0042, since that is the address of MyLabel. The `Bytes to relocate` box should be filled in with $02, since MyLabel|2 uses absolute addressing, which is two bytes on the 65816. Finally, the `Shift count` box should contain $02, since the address needs to be shifted by two bits for the LDA instruction.

Added relocation information is not saved in the templates for OMF files, so do not use this command with OMF files. The exception to this rule is for bank-relative OMF files, generated by version 1.2.2 or later of the ORCA system linker. You can add relocation information to the templates generated for these types of files.

Remove... (-)

The `Remove...` command allows you to remove definitions for labels, comments, constants, and directives from the current disassembly. It brings up the dialog box:



At the top of the dialog box are two line-edit boxes which ask for a starting and ending address in which to apply the removal of definitions. The addresses should be given as hexadecimal values specifying offsets from the beginning of the segment. The definitions will be removed from the current segment, within the range specified.

Beneath the address boxes are check boxes for labels, constants, comments, and directives. Check the boxes as appropriate.

Clicking on the `Remove` button causes the removal of the definitions to be performed.

Generate Labels...

This command is used to automatically generate labels throughout the current segment. Labels generate best for OMF files, where the disassembler knows exactly what addresses are directly referenced. The disassembler also generates labels for the targets of relative addressing, such as branch instructions.

Generated label names begin with an L, followed by four or six hexadecimal digits. The digits are derived from the location of the label in the disassembly (its offset from the beginning of

the segment). If the current file has only one segment, the generated labels will contain four digits. If the file contains more than one segment (excluding the Expressload segment), the generated labels will contain 6 digits, the first two being the segment number, and the last four being the offset. If the disassembler finds a reference to another segment with no label, a slash character will appear between the two-digit segment number and the offset.

The command causes an alert box to be displayed which shows the progress of the label generation as a thermometer. For short files with few or no labels, the alert will come up and disappear so quickly that you may not be able to clearly see the box. This is normal, and should not cause you concern. You can abort the generation of labels with ⍁. – that is, hold down the ⍁ key and simultaneously press the period key.

The Find Menu

The `Find` menu allows you to search for strings or labels, "goto" to a specified address, or save your current position on a "stack" to return to later.

Find... (⍁F)

The `Find...` command allows you to search throughout the current segment for a specified string, or to find where a label is first defined. The `Find...` command brings up the `Find` modeless dialog:

At the top of the dialog is a line-edit box where you enter the string or label that is to be found.

Click on the `Find string` button to search for a string, or the `Find label` button to find the first definition of a label.

The `Find next` button is used to search for the next occurrence of the string or label.

The search begins at the current location in the disassembly. If the string or label is found, the disassembly display is scrolled up so that the string or label appears at the top of the display. Subsequent searches will continue from this occurrence downward in the display.

Searching for a string can be quite time-consuming, since the disassembler must disassemble each line of the file to look for the target string. On the other hand, you can search for anything – including opcodes or strings in comments. Searching for a label is very fast, but only labels in the label field are found.

You can cancel the `Find...` command with open-Apple period (⍁.) – hold down the ⍁ key, then press the `.` key.

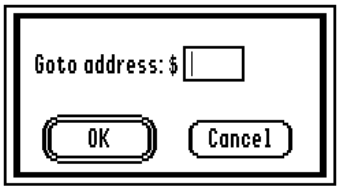
Find Again (Ó L)

The `Find Again` command provides a fast way to search for the next occurrence of the string or label previously defined with the `Find...` command. The search commences at the current location in the disassembly and continues downward in the display.

You can cancel the `Find Again` command with open-Apple period (Ó.) – hold down the Ó key, then press the **.** key.

Goto... (Ó G)

The `Goto...` command allows you to move to a specified address in the current segment using the `Goto` dialog:



The address you should enter in the `Goto address` box is the offset, given as a hexadecimal number, from the beginning of the segment to the desired location.

Click on the `OK` button to perform the goto operation, or click on the `Cancel` button to cancel the `Goto...` command.

Save Location (Ó( )

The `Save Location` command saves the current segment and location within the segment on a "location stack" so you can return to this location later by using the `Restore Location` command. As many as ten locations can be saved during one disassembly session. The locations are stacked, so that the first location to be restored will be the last location that was saved. In the event that you attempt to stack more than ten locations, the oldest locations are lost.

Restore Location (Ó))

The `Restore Location` command restores the most recently saved location (from the `Save Location` command). When you issue this command, the disassembly display will be scrolled so that the restored location is displayed at the top of the window, and the current location within the disassembly is set to this location. The location information is removed from the top of the location stack. Up to ten saved locations can be pending at any one time.

The Mode Menu

The Mode menu lets you change some general aspects of the disassembly. For example, you can choose to display all Apple IIGS toolbox calls as macros or simply with the name of each toolbox call in a comment; you can specify upper- or lower-case display of opcodes; you can specify whether to view the current segment as a series of hexadecimal values instead of as assembly-language source code; you can also change the current accumulator and index register sizes.

View Hex (H)

The `View hex` command switches between displaying the current segment in hexadecimal machine code or as assembly-language source code. The hex mode display shows each byte in the segment as both a two-digit hexadecimal number representing the ASCII value of the byte, and as an ASCII character. Characters that cannot be displayed are replaced by a period (.).

Toggle Memory Size ([ )

The `Toggle memory size` command toggles the accumulator register's width between 8 and 16 bits starting at the current top line of the disassembly.

This command is generally only used when the disassembler is out of sync for some reason; to make a permanent change to the register state that the disassembler will remember and reflect in the source code it generates, you should use the `Define Directive...` command to put in LONG A or SHORT A directives.

Toggle Index Size ( ])

The `Toggle index size` command toggles the index (X/Y) register width between 8 and 16 bits starting at the current top line of the disassembly.

This command is generally only used when the disassembler is out of sync for some reason; to make a permanent change to the register state that the disassembler will remember and reflect in the source code it generates, you should use the `Define Directive...` command to put in LONG I or SHORT I directives.

Toolbox Macros

The `Toolbox macros` command specifies whether Apple IIGS toolbox calls will be displayed as standard ORCA/M and APW macro calls (the default), or as the actual code used to make the call, with the toolbox command's name in a comment next to the call. The `Toolbox macros` setting will be saved in the DISASM.CONFIG file.

Lowercase Opcodes

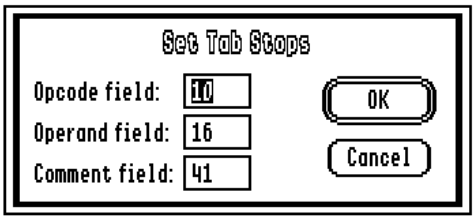
The `Lowercase opcodes` command toggles display of opcodes between upper-case and lower-case (the default). The `Lowercase opcodes` setting will be saved in the DISASM.CONFIG file.

Semicolons Before Comments

The `Semicolons before comments` command toggles display of comments between those preceded by semicolons and those without semicolons (the default). The `Semicolons` setting will be saved in the DISASM.CONFIG file.

Tab Stops...

The `Tab stops...` command lets you specify the placement of assembly-language source-code fields. It brings up the dialog box:



The `Tab stops` line-edit boxes are decimal values, and are the distance, in characters, from the beginning of the label field to the beginning of the specified field. The tab stops will be saved in the DISASM.CONFIG file.

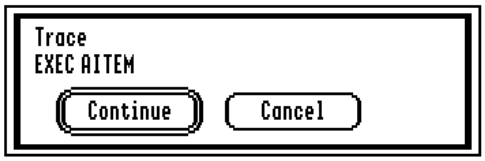
The Scripts Menu

The `Scripts` menu allows you to execute ORCA/Disassembler scripts as well as load, edit, and save script files. ORCA/Disassembler scripts are discussed in detail in the next chapter.

At the bottom of this menu is a list of scripts that can be executed directly by selecting them as you would any other menu item. See the description of the `Edit Script Menu` command, below, for information about tailoring this menu to suit your needs.

Trace Scripts

If the `Trace Scripts` command is used before you execute a script (see `Execute Script...`, below), this dialog will be present during the execution of the script:

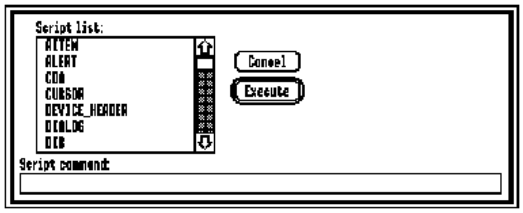


At the top of the box is the name of the script file being executed. Beneath the name is a button marked `Continue`, used to execute the next line in the script, and a button labeled `Cancel`, used to cancel the script.

The dialog box shows the contents of each line as the script executes, and waits for you to click the `Continue` or `Cancel` button before executing the next line in the script.

Execute Script... (⌘\)

The `Execute script...` command allows you to execute scripts using the `Execute Script` dialog:



On the left-hand side of the dialog box is a list of available scripts, obtained from the currently loaded script file. If you do not specifically load in a script file with the `Load Script File...` command, then the scripts in the file DISASM.SCRIPTS are displayed.

To execute a script, use the mouse to select a script. After selection, the line-edit box labeled `Script command` will be filled in with a line such as:

```
EXEC <scriptName> |
```

where `EXEC` is the script command used to execute scripts, `scriptName` is the name of the selected script to execute, and the vertical bar (|) is the insertion point for the line-edit box. If the disassembly window has selected text, the disassembler will also place the address of the first selected byte on the line as a parameter. You would normally begin typing at the insertion point to pass the script any parameters.

The `Cancel` button is used to cancel the `Execute script...` command, while the `Execute` button is used to begin script execution.

If you have selected some text prior to issuing the `Execute script...` command, the disassembler will fill in the line-edit box with default parameters based on the text selected.

You can also execute a script without selecting a script from the script list. In this case you would enter into the `Script command` line-edit box

```
EXEC <scriptName> <parameters>
```

where EXEC, scriptName, and parameters are as described above. Note: Script names are case-sensitive – enter a script name exactly as it appears in the script list.

The Script line-edit box can also be used to directly execute any script language command, with the exception of GO, IF, and ON. Simply type in the command, and then click the Execute button or press RETURN.

Edit Script File

The Edit script file command allows you to edit the current script file. The command brings up a window containing the current script file. Any changes you make in this file are automatically used the next time you execute a script. You can save any changes you might make to the script file with the Save script file as command.

Load Script File...

The Load script file... command allows you to load a new script file. The new script file will replace the script file currently in use. The command brings up a standard Open dialog box, similar to that described earlier in this chapter for the Open command in this chapter. Prior to version 1.2 of ORCA/Disassembler, the types of script files that could be loaded were text files. Starting with version 1.2, script files have a filetype of $B0 (source file) and an auxtype of $0116 (new language type of "Disassembly script;" the high byte of $01 indicates a Byte Works disassembly script). Only these two types of script files can be loaded by the disassembler.

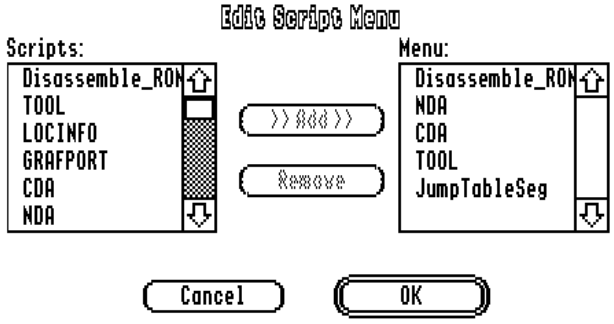
Save Script File As...

The Save script file as... command allows you to save a script file to disk. The command brings up a standard Save dialog box, similar to that described in the Save Template As... command earlier in this chapter.

Edit Script Menu

The Edit script menu command allows you to add up to nine scripts to the Scripts menu. The scripts can then be executed directly by selecting them from the menu. The menu's scripts are given key equivalents of ⌘1 through ⌘9, based on the order in which they are placed in the menu.

The command brings up a dialog similar to that depicted below:



On the left-hand side of the dialog is a list containing the names of all of the scripts in the currently loaded script file. On the right side of the dialog is a list of the scripts that are currently available from the `Scripts` menu. Between the two lists are two buttons, labeled `Add` and `Remove`. To add a script to the menu, select a script from the `Scripts` list, then click the `Add` button. (The `Add` button will not be selectable if the `Menu` list already contains its maximum of nine scripts.) To remove a script from the menu, select the script from the `Menu` list, then click the `Remove` button. (The `Remove` button will be unselectable if the `Menu` list is empty, or if a script is selected from the `Scripts` list rather than the `Menu` list.)

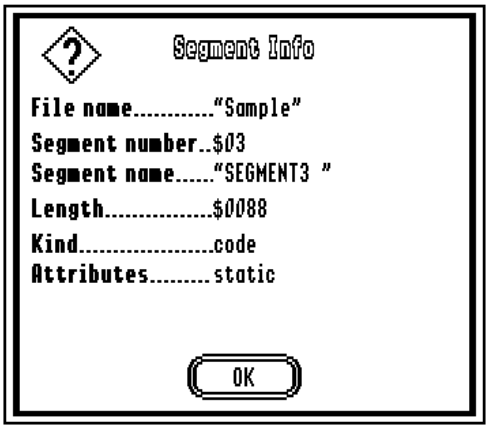
Click the `Cancel` button to exit the dialog without altering the `Scripts` menu. Click `OK` to complete editing of the `Scripts` menu.

The Segments Menu

The `Segments` menu allows you to select a segment of the current file, and allows you to get information about the current segment. This menu only appears after you have opened a file to disassemble.

Segment Info (Ⓐ I)

The `Segment info` command brings up an alert box containing information about the current segment. The dialog will be similar to this one:



The segment information displayed in the box is derived from the object module format header created by the compiler/assembler/linker used. The segment kind can be code, data, jump table, pathname, library dictionary, initialization, absolute bank, direct page, or stack. The segment attributes can be static or dynamic, private, and/or position independent.

Segment information about code resources is also available. The display includes the resource's filename, resource type, resource ID, segment name, length, segment kind, and segment attributes.

Segment Names

Each of the segments in the file loaded (one if BIN or SYS) has an item in this menu. Selecting the menu item corresponding to the segment will make that segment the current segment.

Several of the disassembler's commands refer to segments by number; the segments are numbered starting at 1, based on their appearance in the `Segments` menu, from top to bottom.

Quick Segment Selection

There is a faster method for making a given segment the current one – simply press OPTION-X, where X is the number of the segment to select. For example, if the Segments menu showed these segments:

~ExpressLoad
blank segment
otherSegment

the keystroke OPTION-3 (hold down the OPTION key and simultaneously press the 3 key) would select otherSegment.

Chapter 4
ORCA/Disassembler Script Command Language

Scripts

ORCA/Disassembler comes with its own script language. A script is a series of disassembler commands used to automate the disassembly process. Scripts will most often be used from the text-based disassembler; you will note that many of the commands described in this chapter are available as menu commands in the desktop version of the disassembler.

When the disassembler is first run, it looks for a text file named DISASM.SCRIPTS in its directory (the folder containing the ORCA/Disassembler program). This file contains code for several disassembler scripts. You can edit this file from the desktop disassembler, adding your own scripts and changing the ones that are provided. You can also create any number of script files, loading them in as needed during a disassembly session.

The disassembler's script files use a new language type. To add the language to ORCA/M or APW, add this line to the SYSCMND file:

```
SCRIPT      L           278         ORCA/Disassembler script file
```

Script files have a filetype of $00B0, and an auxtype of $00000116 (278 decimal).

Scripts are executed from the `Execute Script` dialog in the desktop disassembler or directly from the command line in the text-based disassembler by use of the EXEC command (or "\"):

```
EXEC scriptName[,argument]
\scriptName[,argument]
```

Each script in the file is delimited by the SCRIPT and ENDS commands:

```
SCRIPT scriptName

: (script statements)

ENDS
```

Any disassembler script language statement can be used in the script.

Execution of a script begins at the line after the SCRIPT statement, and continues until it reaches the ENDS statement.

Scripts can call other scripts using the EXEC command, up to 16 call levels deep.

Variables and Expressions

Scripts can make use of variables and expressions. Disassembler variables begin with the at-sign character (@), followed by as many as 18 letters, digits, and underscores (_). Variables can hold a one-word integer value.

The following are reserved standard variables. Their values can be used, but not set:

variable	explanation
@ARG	This variable contains the value of the argument passed to the script.
@BUTTON	This variable contains the user's response to a `NOTE`, `INPUT`, or `ALERT` script command (0 = `RETURN`; 1 = `ESCAPE`). (In the desktop version of the disassembler, 0 is used for the OK button, while 1 is used for the CANCEL button.)
@CURSEG	This variable contains the current segment number.
@INPVAL	This variable contains a 4-digit hexadecimal value, the user's response to an `INPUT` script command.
@KEY	This causes the computer to wait for a keystroke and pass its value back to the script. @KEY only functions in the text version of the disassembler.
@NUMSEGS	This variable contains the number of segments in the current file.
@SEGKIND	This variable contains the kind of the current segment, as defined in the file's OMF header.
@SEGLEN	This variable contains the length of the current segment.
@TOPLINE	This variable contains the address of the top line currently showing in the disassembly display.

An expression is made up of numbers (integers) and variables with operators between them, in a form similar to that used by most programming languages. For example:

```
@VAR1=@VAR2+2          sets VAR1 to the value of VAR2 plus 2
```

The following is a list of permissible operators:

arithmetic operations

+	add
-	subtract
*	multiply
/	divide
%	modulo division (remainder)

logical operations

<	less than
>	greater than
=	equal to
<=	less than or equal to
=<	less than or equal to
>=	greater than or equal to
=>	greater than or equal to
<>	not equal to

bitwise operations

<<	shift left
>>	shift right
~	bitwise NOT
&	bitwise AND
|	bitwise OR
^	bitwise XOR

other operations

(expr)	evaluates the expression in parentheses before continuing
[expr]	evaluates to the word value at the address `expr` in the current segment (similar to the PEEK command in Applesoft).

The result of an expression is an integer. The result of a logical expression is either true (-1 or $FFFF) or false (0). Precedence is left-to-right, except that parentheses can be used to override the default precedence. Shift operations are by bits.

The Script Language Commands

In the description of the disassembler language commands, the square brackets ([ ]) denote optional parameters. The ** denotes commands which can only be executed from the text-based version of the disassembler. Unless stated otherwise, the address parameters for the commands are hexadecimal values which are offsets from the beginning of the segment to the object referenced in the command.

With the exception of GO, IF, and ON, all of the commands described below are valid commands that can be entered on the command line in the text-based disassembler. Thus, this chapter describes the command language of the text-based disassembler.

ADDREL offset[,target[,size[,shift]]]

Adds a new relocation entry for a specified address. This command will not normally be used when disassembling OMF files, but may be used when disassembling BIN files or bank relative files.

The optional fields default to `shift` count = 0; `size` = 2 (1 for binary files); `target` = constant value at [`offset`]. This allows you to specify just the offset in most cases.

The effect of `ADDREL` is to change an operand from a constant to a label. For example, when disassembling the following from a binary or bank-relative file:

```
$1000    LABEL  PEA    LABEL|-16
$1003           PEA    $1000
```

the disassembler assumes that all PEA operands are constants (unless they have an associated relocation record), and so assumes the $1000 is a constant. Issuing the command:

```
ADDREL 1000
```

creates a relocation record for the operand, telling the disassembler that the operand is not a constant, and that it should disassemble the example as:

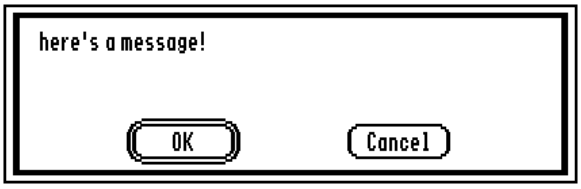
```
$1000    LABEL  PEA    LABEL|-16
$1003           PEA    LABEL
```

Added relocation information is not saved in the templates for OMF files (except for those with bank-relative segments), so do not use this command with OMF files.

ALERT ["string"][variable]

Echoes the string (delimited by single or double quotes) and/or the value of a variable to the screen. The `ALERT` command awaits user input, accepting either a `RETURN` or `ESCAPE`. Upon completion, the standard variable `@BUTTON` contains the user's response (0 = `RETURN`; 1 = `ESCAPE`).

In the desktop version of ORCA/Disassembler, execution of the ALERT command causes a simple alert to be displayed, with the message at the top of the alert window and OK and Cancel buttons along the bottom:



Clicking the OK button corresponds to pressing the RETURN key in the text version, while selecting the Cancel button corresponds to pressing ESCAPE.

ASSUME address,dataBank

Used with non-OMF files only. Causes the disassembler to "assume" a particular data bank for all short (absolute) operands, starting at the specified address. Use 'K' for the data bank to specify the current segment.

The data bank information is not saved in the templates for OMF files, so do not use ASSUME with OMF files.

**CATALOG [pathname]

Shows a list of the files on a disk or subdirectory.

CLEAR

Deletes all the variables currently in use.

CODE startAddress[.endAddress]

Removes any constant definitions for the specified address range. If endAddress is omitted, then only the constant at startAddress is affected.

COMMENT address,commentString

Attaches a comment to the specified address. Comments are limited to 40 characters.

CREATE pathname

Creates a source file from the current segment. The `CREATE` command automatically generates USING and GEQU directives, and splits the source file into approximately 48 KB chunks on START/END/DATA boundaries where possible.

See the description of the `Create source file...` command in the previous chapter for more information about this command.

You can cancel the `CREATE` command with open-Apple period (o.) – hold down the o key, then press the `.` key.

CSTR address

Defines a C string (zero-terminated string) starting at the current address, and extending to the first zero byte past the address or the end of the segment, whichever comes first.

DATA address[,label]

Begins a DATA area at the address specified, giving it the name specified in the label field (if given).

DC startAddress[.endAddress][,constType]

Defines an ORCA/APW assembler DC constant for the specified address range. If `endAddress` is omitted, then the `endAddress` is deduced from the `constType` specified. If the `constType` is omitted, the type DC I is assumed. The constant types allowed are:

A[n[-]]	address
B	binary
C	character constant
D	8-byte double-precision SANE floating-point type
E	extended 10-byte SANE floating-point type
F	4-byte single-precision SANE floating point type
H	hexadecimal constant
I[n[-]]	integer

[n] specifies the size of the address or integer, and ranges from one to eight bytes.

[-] indicates that the intended constant value is one less than the actual value of the constant in the code. See the description of the `-1` parameter under the `Define constant...` command in the previous chapter for more information.

DCOMMENT startAddress[.endAddress]

Removes all comments from the address range specified. If `endAddress` is omitted, then only the line at `startAddress` is affected.

DLABEL startAddress[.endAddress]

Removes all labels, register size declarations, and START/DATA directives from the specified address range. If `endAddress` is omitted, then only the line at `startAddress` is affected.

DPAGE dpageRef,address,label

Attaches a label to the direct-page address specified (the address given must range between $00 and $FF), in the direct page area referenced by `dpageRef`. The reference number must be between $0000 and $FFFF. The direct page reference number is defined by the `SETDPAGE` command, described later in this chapter.

DS startAddress.endAddress

Defines an ORCA/APW assembler DS directive for the address range specified.

DW address

Defines a Pascal type string (leading length byte) starting at the specified address.

**ECHO ["string"][variable]

Echoes the string (delimited by single or double quotes) and/or the value of a variable to the screen.

ENTRY address[,label]

Attaches an ORCA/APW assembler ENTRY directive to the address specified, placing a label name in the label field if a label name is given.

EXEC scriptName

Executes the specified named script from the current script file. (The EXEC command can be replaced with a single backslash, so that the command can also be written: `\scriptName`).

FIND string

Searches starting from the cursor for the specified string, displaying it at the top of the disassembly window if found, and beeping if not. To find the next occurrence, use the `@-L` command.

You can cancel the FIND command with open-Apple period (.) – hold down the key, then press the . key.

GEN

Generates labels for all relocated addresses and addresses referenced by branching instructions. For binary files, the GEN command assumes that all immediate values apply to constant data, and are therefore not addresses. You can use the ADDREL command to add relocation information to non-OMF files.

GO sequenceSymbol

This command can only be used in a script. It unconditionally transfers program control to sequenceSymbol. A sequence symbol is a line containing only a period followed by a label. For example:

```
        GO     .AWAY
        . . .
   .AWAY
```

GOTO [address]

Starts listing the disassembly starting at the address specified. If no address is given, it continues the listing from the last line on the screen. GOTO is equivalent to the LIST command.

**HELP

Prints a list of the most-often used disassembly commands on the screen.

IF expression,sequenceSymbol

This command can only be used in a script. The expression is evaluated; if the result is TRUE (non-zero), program control is transferred to sequenceSymbol; otherwise execution continues at the statement following the IF statement in the script. A sequence symbol is a line containing only a period followed by a label. For example:

```
        IF     @ARG,.TRUE
        . . .
   .TRUE
```

INPUT ["string"][variable]

Echoes the string (delimited by single or double quotes) and/or the value of a variable to the screen. The INPUT command awaits user input, accepting a 4-digit hexadecimal value, terminated with either a RETURN or ESCAPE. Upon completion, the standard variable @INPVAL contains the number, while @BUTTON contains the terminating code (0 = RETURN; 1 = ESCAPE).

LABEL address,label

Attaches a label to the address specified. If the label contains the "+" or "-" characters, it is not actually displayed in the disassembly; the disassembler assumes that you are specifying an offset from the actual label.

LIST [address]

Starts listing the disassembly beginning at the address specified. If no address is given, it continues listing from the last line on the screen. The LIST [address] command may also be issued with [address]L for consistency with the Apple IIGS' built-in monitor program.

LOAD pathname

Loads the specified file from the disk for disassembling. ORCA/Disassembler can disassemble any type of code file – OMF (with the exception of LIB files), BIN, ROM, and SYS.

LONGA address

Attaches an ORCA/APW LONGA ON directive to the address specified.

LONGI address

Attaches an ORCA/APW LONGI ON directive to the address specified.

NEW

Deletes all of the current file's label information from memory.

NOTE ["string"][variable]

Echoes the string (delimited by single or double quotes) and/or the value of a variable to the screen. The NOTE command awaits user input, accepting either a RETURN or ESCAPE. Upon completion, the standard variable @BUTTON contains the user's response (0 = RETURN; 1 = ESCAPE).

In the desktop version of ORCA/Disassembler, execution of the NOTE command causes a simple alert to be displayed, with the message at the top of the alert window and an OK button along the bottom:

Clicking the OK button corresponds to pressing the RETURN key in the text version. There is no correlation with ESCAPE in the desktop version.

ON expression,sequenceSymbol[,sequenceSymbol...]

This command can only be used in a script. The expression is evaluated. If the result is zero, control is transferred to the first sequenceSymbol. If the result is one, control transfers to the second sequenceSymbol, and so on. If there are not enough labels corresponding to the final value of expression, execution continues with the next line in the script. A sequence symbol is a line containing only a period followed by a label. For example:

```
        ON    @ARG,.ZERO,.ONE
        . . .
.ZERO
        . . .
.ONE
```

OPCASE

Toggles the case (upper-case or lower-case) of the opcodes in the disassembly display.

PREFIX prefix

Sets the current GS/OS prefix. The prefix parameter can be any valid prefix recognized by the APW/ORCA shells, and can include prefix numbers, device numbers, and full or partial pathnames.

This command is not available in the desktop version of the disassembler.

PRINT [startAddress.endAddress]

Prints a range of the disassembly to the printer. If the range is omitted, it prints the contents of the current segment.

You can cancel the `PRINT` command with open-Apple period (**.**) – hold down the key, then press the **.** key.

QUIT

Quits this session with the disassembler, returning to the launching program.

RECT address

Defines a rectangle starting at the specified address. A rectangle is defined by the toolbox as a series of four integers (top, bottom, left, right).

RLOAD pathname[,resource type]

Loads into the disassembler all code resources of the `resource type` specified, for the `pathname` containing a resource fork. If the `resource type` is omitted, $8017 ("generic" code resource) is assumed.

ROM

Begins disassembly of the Apple IIGS built-in ROM. It creates pseudo-segments for up to 256KB of ROM, plus the "slow RAM" (banks $E0 and $E1). The following table indicates which banks of memory are mapped into which pseudo-segments:

memory bank	segment
ROM $FF	$01
ROM $FE	$02
ROM $FD	$03
ROM $FC	$04
RAM $E1	$05
RAM $E0	$06

Banks $FC and $FD are only present in ROM version 03.

SEG segNum

Switches to another segment in this file. The `segNum` parameter should be specified as a hexadecimal value.

SEMICOLONS

Toggles the display of semicolons before comments.

SETDPAGE address,dpageRef

Changes the direct page currently in use to that given in `dpageRef`, starting at `address`. DpageRef $0000 is used for the default direct page area. We recommend that you set `dpageRef` for other direct pages to the address at the which the direct page is first used.

SHORTA address

Attaches an ORCA/APW assembler LONGA OFF directive to the address specified.

SHORTI address

Attaches an ORCA/APW assembler LONGI OFF directive to the address specified.

START address[,label]

Attaches an ORCA/APW assembler START directive to the `address` specified, naming the new code segment if a `label` name is given.

TABS opcodeTab,[operandTab,[commentTAb]]

Allows you to define tab stops in the source file for the opcode, operand, and comment fields. Each parameter should be specified as a decimal value.

TLOAD pathname

Loads a template file from the specified pathname. Templates are discussed in the previous chapter.

TOOLMACS

Toggles toolbox call name display between toolbox macros and source code with comments.

TSAVE pathname

Saves the current template information to disk under the specified pathname. See the `TLOAD` command in this chapter.

.label

Specifies a line as a sequence label for the `GO`, `IF`, and `ON` statements. These labels are local to each script.

Chapter 5
The Text-Based Disassembler

The text-based version of ORCA/Disassembler was designed around a different interface philosophy from the desktop version. While the desktop disassembler is based on windows and pull-down menus, the text disassembler is centered around the command line. All commands are entered by name on the command line. While the desktop disassembler can display any number of segments at one time in separate windows, the text version only displays the current segment at any one time. Some users – particularly those used to command-driven utilities – may find that although the text version is more difficult to learn, it may be faster and better suited to their needs than the desktop version.

The previous chapter presents the commands available in the text-based disassembler.

Differences Between the Text-Based and Desktop Versions

Most of the disassembly commands are available in both the text and desktop versions of the program, but the presentation of the commands is different. The commands that are available in the text version that are not (directly) available in the desktop version include HELP, CATALOG, ECHO, ROM, and PREFIX. The HELP command is superfluous in the desktop, since all of the commands can be seen by pulling down menus. Most of the information given by the CATALOG command can be obtained with one of the SAVE AS... commands, although use of SAVE AS... is not as straight-forward as is CATALOG. The ECHO command is not available in the desktop because there is no information area set aside for it. ROM code can be disassembled under the desktop, but it requires execution of a script which contains the ROM script command. The functionality of the PREFIX command, like the CATALOG command, can be simulated under the desktop by using any of the `File` menu commands to change the current prefix.

The commands that are available in the desktop version of ORCA/Disassembler that are not duplicated in the text version include all of the `Scripts` menu commands except for `Execute Script...`, and `Edit Direct Page...` The `Scripts` menu commands can be simulated by creating new script files and/or modifying the script file named DISASM.SCRIPTS which comes with the disassembler package. In order to change the current script file, you will need to exit the text-based disassembler, replace the current script file with the desired script file, and then re-execute the disassembler. Determining whether the file currently being disassembled has an expressload segment is not as straight-forward in the text version as it is in the desktop. An expressload segment is generally the first segment in the disassembly. If the file you load to disassemble under the text version starts at segment number two, rather than one, you will know that the file begins with an expressload segment. The direct page areas defined under the text version of the disassembler cannot be directly edited; you can only add or remove location definitions.

Running the Text-Based Disassembler

The text-based disassembler is executed by entering its name at the shell prompt, just like running any other ORCA/M or APW utility:

```
DISASM [filename [template_name]]
```

`Filename` and `template_name` are optional parameters. `Filename` is the pathname of a file to automatically load for disassembling, and `template_name` is the pathname of the template file for `filename`. Note: It is assumed here that you have installed the text version of ORCA/Disassembler as a utility into your ORCA/APW system. Installation of ORCA/Disassembler is covered in Chapter 1.

The Disassembler Screen

The text disassembler's main screen is be divided into four areas:

1. The information bar, at the top of the screen, gives the name of the program being disassembled, and the name and size of the current segment.

2. The disassembly area, in the middle of the screen, which displays 19 disassembled lines of the current segment.

3. The message area, under the disassembly window. This the the area where ORCA/Disassembler will show any status or error messages, and where any ECHOs, NOTEs, ALERTs and INPUTs from executed scripts will appear.

4. The command line, at the bottom of the screen, which is where you type commands to the disassembler.

The Line Editor

All commands to the text-based disassembler are entered on the command-line. The available line-editing commands are listed in the table below.

command	command name and effect
LEFT-ARROW	Cursor left – the cursor will move to the left on the command line.
RIGHT-ARROW	Cursor right – the cursor will move to the right until it reaches the end of what has already been typed.
⌘> or ⌘.	End of line – the cursor will move to the end of was has been already typed on the command line.
⌘< or ⌘,	Start of line – the cursor will move to the start of the command line.

DELETE	Delete character left – deletes the character to the left of the cursor, moving the cursor to the left.
CONTROL-Y	Delete to end of line – deletes characters from the cursor to the end of the line.
CONTROL-E	Toggle insert mode – allows characters to be inserted into the command line. The insert cursor appears as a blinking underscore.
CLEAR	Clear command line – removes all characters from the command line.
CONTROL-X	Clear command line – removes all characters from the command line.
RETURN	Execute command – issue the current command to the disassembler, and append the command to the list of most recent sixteen commands.
ENTER	Execute command – issue the current command to the disassembler, and append the command to the list of most recent sixteen commands.

Scrolling the Disassembly

The disassembly display may be scrolled both forward and backward using the following commands:

command	command name and effect
⌘-DOWN-ARROW	Page down – scroll the disassembly down one page (19 lines).
⌘-UP-ARROW	Page up – scroll the disassembly up one page (19 lines).
DOWN-ARROW	Scroll down – scroll the disassembly down one line.
UP-ARROW	Scroll up – scroll the disassembly up one line.

Scrolling Through Commands (Command History)

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

Other Command Line Commands

There are also a number of other commands that may be entered directly at the command line, without disturbing the command being edited.

command	command name and effect
ÅL	Find next – find the next occurrence of the string entered when the `FIND` command was last issued.
ÅH	Toggle hex mode – toggles the display between a hex dump of the segment and an assembly-language source listing.
ÅM	Toggle memory mode – toggles the A register/memory width between 8 and 16 bits for the current (top of screen) line of the disassembly.
ÅX	Toggle index mode – toggles the index (X/Y) register width between 8 and 16 bits for the current (top of screen) line of the disassembly.
?	Help – displays a screen listing some of the ORCA/Disassembler commands that are available.
? commandName	Provides information about the command specified in the commandName parameter.

Appendix A
Disassembling "Special" File Types

Binary Files

Disassembling binary files is very different from disassembling OMF files. Binary files contain no relocation or label information, making it much harder for the disassembler to tell the difference between code and data.

The place this becomes most apparent is in a statement such as:

```
PEA    #Label
```

If this statement occurred in an OMF file, the disassembler would know that the operand refers to a relocatable label, rather than an integer constant. However, in a binary file, the disassembler cannot know. Therefore, all immediate mode operands are assumed to be integer constants, rather that labels, since that is the usual case.

To handle this problem, you can either create a relocation record for the statement (using the `ADDREL` command), or make a note (perhaps a comment) to change the resulting source file after creating it.

The load address for a binary file comes from the file's auxType, unless it is a SYS file, in which case the load address is assumed to be $2000.

Another concern when disassembling non-OMF files is the data bank, i.e., the target of the B register. For ProDOS 8 programs, this will usually be the same as the code bank – which is the default. Some programs, however, change the data bank register. In order for the disassembler to find the correct label for an address, it has to know which bank the address is in. This is the purpose of the `ASSUME` command. This command tells the disassembler to "assume" that the data bank register is pointing to a specified bank. For example, the disassembler may encounter the following sequence in bank $00:

```
        :
0100    PH2     #$E1E1
0103    PLB
0104    PLB
0105    LDA     $0100
        :
```

In this case, the data bank register points to bank $E1, so the LDA's actual target address is $E10100, not $000100. An `ASSUME 100,E1` would tell the disassembler that the data bank is pointing to bank $E1 starting at $0100 in the segment.

Use `ASSUME (address),K` to change the data bank back to the current segment. (K is the name of the code bank register.)

OBJ (Object) Files

Disassembling OBJ files is very different from disassembling other types of OMF files. Just as OMF files contain a lot more information than binary files, OBJ files contain a lot more information that other types of OMF files. The exact information contained in the OBJ file depends on what assembler or compiler generated the file. Generally, OBJ files include the names of all global labels and, often, constant definitions.

Typically, after loading an OBJ file, you will find that all of the global labels and many constants have already been defined, but the local labels are not. You may also see operands that are complex expressions involving local and global labels, and constants. You should not change the names of any pre-defined labels, since they may be referred to by hard-coded label names in the expressions. The label names within expressions may not change if you rename that label.

Appendix B
DISASM.DATA File Format

General Information

The DISASM.DATA file is a text file containing the information that ORCA/Disassembler needs to know about ToolBox and GS/OS calls, and about the Apple IIGS system globals. The basic layout of the data file is patterned after the data file used by the popular shareware utility "NiftyList." (NiftyList is a CDA that performs a number of functions, including disassembly of toolbox and operating system calls and dumping of toolbox data structures. It is obtainable from its author, Dave Lyons, at: P.O. Box 875, Cupertino, CA 95015-0875. The cost is $15.00.)

The file itself is line-oriented, i.e., each line contains a separate entry. Any line beginning with a semicolon (;) character is ignored, and may be used for comments. Lines beginning with an asterisk (*) separate the sections; anything following the asterisk on the line is ignored, and may be used for comments. All other lines in the file begin with a four-character hexadecimal number, a space or digit, a name or label, or in some instances a comment.

The file is broken up into eight sections of information:

1. ProDOS 8 calls
2. GS/OS, APW/ORCA shell, and ProDOS 16 calls
3. System Toolbox calls
4. User Toolbox calls
5. Bank $E1 globals
6. Bank $E0 globals
7. Bank $01 globals
8. Bank $00 globals

Each section includes a series of entries for that section. The entries must be in numerical order within each section.

Since DISASM.DATA is a text file, you can display its contents, edit it, or print it.

Operating System and Shell Calls

The entries in the first two sections – the operating system and shell calls – are very similar, and look as follows:

```
2010      Open
A     B   C
```

Field "A" is the call number. It must always be four digits long (use leading zeroes if necessary). Field "B" is a required space. Field "C" is the name of the call, as it appears in the manual describing the call.

Toolbox Calls

The entries for the system and user Toolbox sections are very similar, and look as follows:

```
090E      NewWindow
A     B   C
```

Field "A" is the Toolbox call number. It must always be four digits long (use leading zeroes if necessary). Field "B" is a required space. Field "C" is the name of the Toolbox call, as it appears in the *Toolbox Reference* manuals.

System Globals

The entries for the last four sections – system globals – are very similar, and look as follows:

```
FC503SET_SYS_SPEED            control processor speed
A   BC                        D
```

Field "A" is the address of the global. It must always be four digits long (use leading zeroes if necessary). Field "B" indicates the size of the global, minus one. For example, if the global is a two-byte (one word) value, field "B" should be a "1". Likewise, if the global is a four-byte (two word) value, field "B" should be a "3". If the global is one byte long, field "B" can be either a "0" or a space. Field "C" is the name of the global. Field "D" is a comment briefly explaining the global.

Appendix C
Description of Sample Scripts

This appendix describes the scripts that are included in the ORCA/Disassembler package. The scripts are contained in the file named DISASM.SCRIPTS.

All scripts except CDA, DEVICE_HEADER, Disassemble_ROMs, FST_HEADER, JumpTableSeg, NDA and TOOL take a parameter that gives the start of the data structure defined by the script.

AITEM: Creates an ALERT ITEM template.

ALERT: Creates an ALERT template.

CDA: Creates a CDA header.

CURSOR: Creates a cursor image.

DEVICE_HEADER: Creates a GS/OS device driver header.

DIALOG: Creates a DIALOG template.

DIB: Creates a GS/OS device driver device information block.

Disassemble_ROMs: Allows the desktop user to easily disassemble the Apple IIGS ROM.

EVENTREC: Creates an event record.

FONT: Creates a font definition.

FORMAT_OPTIONS: Creates a GS/OS device driver format options table.

FST_HEADER: Creates a GS/OS FST header.

GRAFPORT: Creates a QuickDraw II GrafPort record.

GSOS_BeginSession: Creates a BeginSessionGS parameter block.

GSOS_BindInt: Creates a BindIntGS parameter block.

GSOS_Buffer: Creates a GS/OS result buffer.

GSOS_ChangePath: Creates a ChangePathGS parameter block.

GSOS_ClearBackup: Creates a ClearBackupGS parameter block.

GSOS_Close: Creates a CloseGS parameter block.

GSOS_Create: Creates a CreateGS parameter block.

GSOS_DControl: Creates a DControlGS parameter block.

GSOS_Destroy: Creates a DestroyGS parameter block.

GSOS_DInfo: Creates a DInfoGS parameter block.

GSOS_DReadWrite: Creates a DRead/DWriteGS parameter block.

GSOS_DStatus: Creates a DStatusGS parameter block.

GSOS_EndSession: Creates a EndSessionGS parameter block.

GSOS_EraseDisk: Creates a EraseDiskGS parameter block.

GSOS_ExpandPath: Creates a ExpandPathGS parameter block.

GSOS_FileInfo: Creates a Get/SetFileInfoGS parameter block.

GSOS_Flush: Creates a FlushGS parameter block.

GSOS_Format: Creates a FormatGS parameter block.

GSOS_GetBootVol: Creates a GetBootVolGS parameter block.

GSOS_GetDevNumber: Creates a GetDevNumberGS parameter block.

GSOS_GetDirEntry: Creates a GetDirEntryGS parameter block.

GSOS_GetFSTInfo: Creates a GetFSTInfoGS parameter block.

GSOS_GetMarkEOF: Creates a GetMark/EOFGS parameter block.

GSOS_GetName: Creates a GetNameGS parameter block.

GSOS_GetVersion: Creates a GetVersionGS parameter block.

GSOS_Level: Creates a Get/SetLevelGS parameter block.

GSOS_NewLine: Creates a NewLineGS parameter block.

GSOS_Null: Creates a NullGS parameter block.

GSOS_Open: Creates a OpenGS parameter block.

GSOS_OSShutdown: Creates a OSShutdownGS parameter block.

GSOS_Prefix: Creates a Get/SetPrefixGS parameter block.

GSOS_Quit: Creates a QuitGS parameter block.

GSOS_SessionStatus: Creates a SessionStatusGS parameter block.

GSOS_SetMarkEOF: Creates a SetMark/EOFGS parameter block.

GSOS_SysPrefs: Creates a Get/SetSysPrefsGS parameter block.

GSOS_UnbindInt: Creates a UnbindIntGS parameter block.

GSOS_Volume: Creates a VolumeGS parameter block.

JumpTableSeg: Creates a Disassemble a jump table segment.

LISTREC: Creates a list record.

LOCINFO: Creates a Quick Draw II LocInfo record.

NDA: Creates an NDA header.

PAINTPARAM: Creates a PaintPixels parameter block.

P16_ALLOC_INTERRUPT: Creates an ALLOC_INTERRUPT parameter block.

P16_BLOCK: Creates a READ/WRITE_BLOCK parameter block.

P16_CHANGE_PATH: Creates a CHANGE_PATH parameter block.

P16_CLEAR_BACKUP_BIT: Creates a CLEAR_BACKUP_BIT parameter block.

P16_CLOSE: Creates a CLOSE parameter block.

P16_CREATE: Creates a CREATE parameter block.

P16_DEALLOC_INTERRUPT: Creates a DEALLOC_INTERRUPT parameter block.

P16_DESTROY: Creates a DESTROY parameter block.

P16_D_INFO: Creates a D_INFO parameter block.

P16_EOF: Creates a GET/SET_EOF parameter block.

P16_ERASE_DISK: Creates a ERASE_DISK parameter block.

P16_EXPAND_PATH: Creates a EXPAND_PATH parameter block.

P16_FILE_INFO: Creates a GET/SET_FILE_INFO parameter block.

P16_FLUSH: Creates a FLUSH parameter block.

P16_FORMAT: Creates a FORMAT parameter block.

P16_GET_BOOT_VOL: Creates a GET_BOOT_VOL parameter block.

P16_GET_DEV_NUM: Creates a GET_DEV_NUM parameter block.

P16_GET_DIR_ENTRY: Creates a GET_DIR_ENTRY parameter block.

P16_GET_LAST_DEV: Creates a GET_LAST_DEV parameter block.

P16_GET_NAME: Creates a GET_NAME parameter block.

P16_GET_VERSION: Creates a GET_VERSION parameter block.

P16_LEVEL: Creates a GET/SET_LEVEL parameter block.

P16_MARK: Creates a GET/SET_MARK parameter block.

P16_NEWLINE: Creates a NEWLINE parameter block.

P16_OPEN: Creates a OPEN parameter block.

P16_PREFIX: Creates a GET/SET_PREFIX parameter block.

P16_QUIT: Creates a QUIT parameter block.

P16_RW: Creates a READ/WRITE parameter block.

P16_VOLUME: Creates a VOLUME parameter block.

StartStopRec: Creates a StartUpTools start/stop record.

TASKREC: Creates a Window Manager task record.

TASKRECX: Creates a Window Manager extended task record.

TOOL: Creates a toolset header for the current segment.

TOOLTBL: Creates a LoadTools Tool Table.

WINDOW: Creates a NewWindow parameter list.

Appendix D
Error Messages

address error
An invalid address was given as the parameter for a command.

address must be delimited by space or comma
Many commands have an address parameter that must be followed by a space or comma before the next parameter.

align or ds record too large
An ALIGN or DS record of an OBJ file causes the segment to be longer that 64K.

bad bank number
The bank number specified is invalid (>$FF).

bad segment number in labels file
The segment number indicated in the template file does not exist in the file being disassembled. Most likely, the template does not belong to this file.

can't change standard variable's value
The values of standard reserved variables cannot be changed.

comment not found
No comment exists at the address specified.

deferred mode only command
A script language command was issued, but the command can only be used within a script.

direct page page number error
The direct page reference number given was invalid.

direct page number must be delimited by space or comma
When using the DPAGE command, the dpageRef was not followed by a space or comma.

duplicate record for segment
Internal error when loading templates. If this error ever occurs, please contact the Byte Works.

empty SUPER record
The OMF file being loaded contains an empty SUPER record. You will not be able to disassemble the file.

expression too complex
An OBJ file segment contains an expression that is too complex for the disassembler. You will not be able to disassemble the file.

illegal character in comment
The COMMENT command was used with illegal character(s) in the specified comment.

illegal character in label
The LABEL command was used with illegal character(s) in the specified label. Valid characters are A-Z, a-z, 0-9, and _.

illegal record type
The OMF file being loaded contains an illegal record type. You will not be able to disassemble the file.

illegal segment number
The segment number specified does not exist.

illegal template record type
The template file being loaded contains an illegal record.

invalid label file
The file you have attempted to load as a template is not a valid template file.

invalid tab stop specified
An invalid tab stop was specified. Tab stops must be <127 and increasing in value from opcode to operand to comment.

label not found
The label specified in the command does not exist.

missing comment
The COMMENT command was used without specifying a comment.

missing constant type
The DC command was used without specifying a constant type.

missing direct page label
No direct page label was specified in the DPAGE command.

missing label
The LABEL command was used without specifying a label.

missing or improper parameter(s) for relocation entry
Some parameters to the ADDREL command were missing or invalid.

missing ']' in expression

The given expression contains a '[' without a matching ']'.

missing ')' in expression
The given expression contains a '(' without a matching ')'.

missing ',' in IF statement
The IF statement requires a comma between the expression and the target sequence symbol.

missing ',' in ON statement
The ON statement requires commas between the expression and the target sequence symbols.

no constant defined at address
No constant exists at the address specified.

no file loaded
Many script language commands require that a file be loaded for disassembly.

no printer/driver found
No printer driver was found. You must use the control panel NDA to select another printer.

no scripts loaded
No script file has been loaded, or the script file has no scripts defined.

relocation dictionary sort error
This is an internal error that you should never see. If this error occurs, please contact the Byte Works.

relocation table unsorted
This is an internal error that you should never see. If this error occurs, please contact the Byte Works.

script call stack overflow
You attempted to nest more that 16 script calls.

script label not found
The specified sequence symbol (used by GO, IF, and ON) was not found.

script not found
The script specified by the EXEC command does not exist.

segment longer than 64K
The OMF file being loaded contains a segment longer that 64K. You will not be able to disassemble the file.

syntax error in script command
The script command contains an unknown statement.

too many relocation records in segment
A segment in the OMF file being loaded contains too many relocation records for the disassembler to handle. You will not be able to disassemble the file.

unknown constant type
The DC command was used with an unrecognized constant type.

unknown OMF file version
The OMF file version is greater that 2, and is incompatible with the current version of the disassembler.

unrecognized SUPER record type
The OMF file being loaded contains a unknown type of SUPER record. You will not be able to disassemble the file.

O

P

Q

R

S