ORCA/M™ 4.1

Macro Assembler

Authors:
Mike Westerfield
Phil Montoya
Patty Westerfield

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 the 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 author of this program are liable or responsible to the purchaser and/or user for loss or damages 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.

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

Use of Libraries - The enclosed subroutine libraries are fully copyrighted works. It is the policy of Byte Works Inc to license these libraries to purchasers of ORCA/M free of charge. Such licenses are generally restricted to including the libraries in binary files, and do not extend to use of the source code. A copy of the program, along with any documentation, and a list of the library subroutines used is required at the time of licensing, and the documentation must give credit for using libraries from ORCA/M. For details, please contact the Byte Works Inc.

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

Dedicated to my Father.
-Mike Westerfield

This program was started in June of 1980 as a summer project. It sort of grew a little in concept over the next few weeks... months... two years. At this point I believe it to be the most complete assembler language development system ever written for a micro computer. I hope that your experiences with it justify that feeling.

I would like to express my thanks to the many people who have helped on this project. I would like to thank CD Osborne, Bruce Benson, Sandy Croushore, Dave Umphress and Rex Fahrquar for proofreading the original User's manual and making many suggestions for improvement. I would also like to thank Ed Martin, Jim Barrie, Bruce Benson and Mike McCracken for their suggestions on improving the assembler. Bruce was of special help, as he patiently beat the concepts of structured programming into my head, and informed me of all those things about programming that nobody bothers to tell a physicist. I would also like to mention my closest friend, Peg Smith. She didn't do anything (specific) to help the assembler along, but she has always wanted to be mentioned in an introduction.

My deepest thanks go to my wife, whose support made things a lot easier throughout the project.

Mike Westerfield
Denver, Colorado
November, 1982

Welcome to the newest version of ORCA/M. A great deal has changed since the original version of ORCA was released in February of 1983. The Byte Works has sprung up around the product, and continues to grow and expand into other areas, and the assembler itself has matured as well. Milestones for ORCA have included the 65816 version, which was the first assembler ever for that chip (and also defined the assembly language syntax used by all 65816 assemblers). This version takes the product towards the UNIX operating system that has become so common.

The number of people who have made contributions to ORCA seems to grow almost daily. Of all of these, one person stands out clearly. While ORCA was published by Hayden, its product manager was David Eyes. He was responsible for a great deal of the editing of the original User's Manual, and shaped the product through its first two years. It was David who was largely responsible for the fact that ORCA was the first assembler to support the 65816. Many of the changes made in the current version are either his ideas, or resulted from long conversations between us.

The Videx 80 column board driver for this version of ORCA was written by Bruce Boeke. I would like to thank him for that contribution.

I would like to welcome Phil Montoya to ORCA. Phil did most of the translation of the editor and operating system as it was converted from DOS to ProDOS, as well as making a number of suggestions in other areas of the program.

Finally, I would like to thank all of you who have contributed your suggestions and pointed out errors. I hope all of the errors have been corrected to your satisfaction, and look forward to hearing from you in the future.

Mike Westerfield
Albuquerque, New Mexico
December, 1984

TABLE OF CONTENTS

Chapter 1: Getting Started 1

How to Use the Manual 1
Booting the System 1
Some Basic Commands 2
A Few Editor Commands 3
Entering a Simple Program 4

Chapter 2: The Command Processor 7

Using Wildcards 7
Required and Optional Parameters 8
Types of Commands 8
The ORCA Monitor Commands 9

Chapter 3: The Editor 11

Introduction 11
Moving Through the File 11
Entering Text 13
Inserting and Deleting Text 13
Moving and Copying Text 14
Search and Replace 14
Other Features 16

Chapter 4: The Fundamental Assembler Directives 17

The Assembly Language Statement 17
Comment Lines 19
KEEP 20
START and END 20
Equates 21
DC and DS 22
Global Labels 25
DATA Areas 26
The APPEND Directive 27

Chapter 5: Advanced Commands 29

Linking to Several Locations 29
Partial Assemblies 30

Subroutine Libraries 32
EXEC Files 33
Device Numbers 34
Redirecting Input and Output 34

Chapter 6: Advanced Assembler Directives 37

COPY 37
Format Control Directives 37
Positioning Code 39
Controlling DC Code 41
The MERR Directive 41
Using the 65C02 41
Using the 65816 or 65802 42

Chapter 7: Using Macros 45

Tools of the Trade 45
The Macro Library 48
Keyword Parameters 49

Chapter 8: Writing Macros 51

MCOPY, MACRO and MEND 51
Basic Parameter Passing 53
Defining Symbolic Parameters 55
Changing and Using Symbolic Parameters 57
String Manipulation 58
Conditional Assembly Branches 59
Attributes 60

Chapter 9: Writing Shell Programs 65

Memory Map 65
Reserving Memory 66
The Console Driver 67
Passing the Command Line 67
Installing a New Utility 68
Installing a Compiler or Editor 69

Chapter 10: Introduction 71

Using The Manual 71
Introduction 71

ORCA.SYSTEM 71
ORCA.HOST 71
MONITOR 72
Utilities 72
System Configuration 72

Chapter 11: The Command Processor 75

Types of Commands 75
Built in Commands 75
Utilities 75
Language Names 75
Program Names 76
Entering Commands 76
File Names 77
Types of Text Files 78
Input and Output Redirection 78
Command Files 79
Command and Utility Reference 79

Chapter 12: The Text Editor 95

Text Entry 95
User Buffers 97
The String Buffers 97
The Character Buffer 97
Information Window 97
Control Key Commands 98
Cursor Movement Keys 98
String Search Commands 99
Miscellaneous Commands 100
Escape Key Commands 102
The Repeat Feature 102
Cursor Movement Commands 103
Text Edit Window Control 103
Insert and Delete 104
Non-keyboard Characters 104
Buffer Commands 105
TAB Stops 106
The Insert Mode 106
Editor Macros 106
Entering Macros 106
Using Macros 107
Defining Tab Stops 107

Chapter 13: The Link Editor 109

Overview 109
The Link Edit Process 109
Object Modules Created by the Assembler 110
Subroutine Selection 110
Link Edit Command Parameters 111
Creating Library Files 112
Output 113
Pass One Output 113
Pass Two Output 113
Global Symbol Table 113
File Length and Error Count 113

Chapter 14: 6502/65C02 Disassembler 115

Introduction 115
Command Descriptions 116
Some Final Comments 120

Chapter 15: Running the Assembler 123

Introduction 123
Assembler Parameters 124
The Assembly Process 126
Pass One 126
Pass Two 126
Controlling the Speed 126
Stopping the Listing 127
Terminal Errors 127
The RESET Key 127
The Assembly Listing 128
Screen Listings 128
Printer Listings 128

Chapter 16: Coding Instructions 131

Types of Source Statements 131
Comment Lines 131
The Blank Line 131
The Characters *, ;, and ! 131
The Period 132
Instructions 132
The Label 132

The Operation Code 132
The Operand Field 133
Instruction Operand Format 133
Expressions 136
The Comment Field 140

Chapter 17: Assembler Directives 141

Introduction 141
Descriptions of Directives 141
ABSADDR OFF Show Absolute Addresses 143
ALIGN Align to a Boundary 143
APPEND Append a Source File 143
COPY Copy a Source File 144
DATA Define Data Segment 144
DC Declare Constant 144
DS Declare Storage 150
EJECT Eject the Page 150
END End Program Segment 150
ENTRY Define Entry Point 151
EQU Equate 151
ERR ON Print Errors 151
EXPAND OFF Expand DC Statements 152
GEN OFF Generate Macro Expansions 152
GEQU Global Equate 152
IEEE ON IEEE Format Numbers 152
INSTIME OFF Show Instruction Times 153
KEEP Keep Object Module 153
LIST ON List Output 153
LONGA ON Accumulator Size Selection 153
LONGI ON Index Register Size Selection 154
MCOPY Copy Macro Library 154
MDROP Drop a Macro Library 155
MEM Reserve Memory 155
MERR Maximum Error Level 155
MLOAD Load a Macro Library 155
MSB OFF Set the Most Significant Bit of Characters 156
OBJ Set Assembly Address 156
OBJEND Cancel OBJ 156
ORG Origin 157
PRINTER OFF Send Output to Printer 158
RENAME Rename Op Codes 158
SETCOM Set Comment Column 159

START Start Subroutine 159
SYMBOL ON Print Symbol Tables 159
TITLE Print Header 160
TRACE OFF Trace Macros 160
USING Using Data Area 160
65C02 OFF Enable 65C02 Code 161
65816 OFF Enable 65816 Code 161

Chapter 18: Macro Language and Conditional Assembly Language 163

Introduction to Macros 163
The Macro File 163
Writing Macro Definitions 163
Symbolic Parameters 165
Positional Parameters 165
Keyword Parameters 168
Subscripting Parameters in Macro Call Statements 169
Explicitly Defined Symbolic Parameters 170
Symbolic Parameter Definition Statements 171
Sequence Symbols 172
Attributes 172
Conditional Assembly and Macro Directives 174
ACTR Assembly Counter 174
AGO Unconditional Branch 175
AIF Conditional Branch 176
AINPUT Assembler Input 177
AMID Assembler Mid String 178
ANOP Assembler No Operation 179
ASEARCH Assembler String Search 179
GBLA Declare Global Arithmetic Symbolic Parameter 180
GBLB Declare Global Boolean Symbolic Parameter 180
GBLC Declare Global Character Symbolic Parameter 181
LCLA Declare Local Arithmetic Symbolic Parameter 181
LCLB Declare Local Boolean Symbolic Parameter 182
LCLC Declare Local Character Symbolic Parameter 182
MACRO Start Macro Definition 183
MEND End Macro Definition 183
MEXIT Exit Macro 183
MNOTE Macro Note 183
SETA Set Arithmetic 184
SETB Set Boolean 184
SETC Set Character 185

Chapter 19: Macro Libraries 187

Addressing Modes 187
Data Types 189
Two Byte Integers 189
Four Byte Integers 190
Eight Byte Integers 190
Character 190
Strings 190
Boolean Variables 191
Memory Usage 191

Chapter 20: Mathematics Macros 193

ABSx Integer Absolute Value 194
ADDx Integer Addition 195
CMPx Integer Compare 197
DIVx Integer Division 199
MODx Integer Modulo Function 200
MULx Integer Multiplication 201
RANx Integer Random Number Generator 202
SIGNx Integer Sign Function 203
SQRTx Integer Square Root 204
SUBx Integer Subtraction 205

Chapter 21: Input and Output Macros 207

ALTCH Select Alternate Character Set 208
BELL Beep the Bell 209
CLEOL Clear to End of Line 210
CLEOS Clear to End of Screen 211
COUT Character Output 212
GETx Variable Input 213
GET_LANG Get Current Language 215
GET_LINFO Get Language Information 216
GOTOXY Position Cursor On Screen 219
HOME Form Feed 220
KEYPRESS Read Keypress 221
NAMEADR Fetch Address of Command Table 222
NORMCH Select Normal Character Set 223
PRBL Print Blanks 224
PUTx Variable Output 225
PUTCR Carriage Return 227
RDKEY Read Keyboard 228

READXY Read Cursor Position 229
SET_LANG Set Language 230
SET_LINFO Set Language Information 231
SIZE Find Screen Size 232

Chapter 22: ProDOS Macros 233

FINDBUFF Find an Unused Memory Buffer 235
RELEASE Release a Memory Buffer 236
RESERVE Reserve a Memory Buffer 237

Chapter 23: Graphics Macros 239

BB Bit Block Definition 240
COLOR Set Pen Color 241
COLORMAP Color Map Enable/Disable 243
DRAWBLOCK Draw a Block 244
DRAWTO Draw a Line 245
FILLSCREEN Fill Screen With a Color 246
FILLSHAPE Fill a Shape 247
FINDXY Find the Pen Position 248
GROUT Graphics Output 249
INITGRAPH Set Up a Screen 251
MOVETO Move the Pen 252
PLOT Plot a Point 253
READXY Read a Point 254
SET_COLOR Define Color Map 255
TEXTOUT Reset Standard Output 256
VIEW Show a Screen 257
VIEWPORT Set Graphics View Port 259
WRITETO Write to a Screen 261

Chapter 24: Miscellaneous Macros 263

ASL2 Two Byte Arithmetic Shift Left 264
BBRx Branch on Bit Reset 265
BBSx Branch on Bit Set 266
BGT Branch on Greater Than 267
BLE Branch on Less Than or Equal 268
BUTTON Read a Game Paddle Button 269
CNVxy Convert x to y 270
CMPW Compare Word 272
DBcn Decrement and Branch 273
DBcn2 Two Byte Decrement and Branch 274

DEC2 Two Byte Decrement 275
DSTR Define String 276
DW Define Word 278
ERROR Flag an Error 279
FLASH Flashing Characters 280
INC2 Two Byte Increment 281
INITSTACK Initialize an Evaluation Stack 282
INVERSE Inverse Characters 283
Jcn Conditional Jumps 284
LA Load Address 285
LM Load Memory 286
LSR2 Two Byte Logical Shift Right 287
MASL Multiple Arithmetic Shift Left 288
MLSR Multiple Logical Shift Right 289
MOVE Move Memory 290
MOVEx Long Memory Moves 291
NORMAL Normal Characters 292
NOTE Play a Note 293
PAGEx Set Display Page 295
PREAD Read a Game Paddle 296
RAM Set the RAM Card Switches 297
RESTORE Restore Registers 298
RMBx Reset Memory Bit 299
SAVE Save Registers 300
SEED Random Number Seed 301
SMBx Set Memory Bit 303
SOFTCALL Soft Reference Call 304

Appendix A: Error Messages 305

Error Levels 305
Recoverable Assembler Errors 305
Terminal Assembler Errors 314
Recoverable Linker Errors 316
Terminal Linker Errors 318

Appendix B: File Formats 321

Overview 321
Text Files 321
Object Modules 321
Binary Files 330

Appendix C: Differences Between ORCA/M 3.5 and ORCA/M 4.0 331

Generalities 331
Changes to Support ProDOS 331
Changes to Support the 65816 332
The 65816 Directive 332
Address Mode Specification 332
Operand Size Specification 333
Word Size Directives 334
Byte Selection Functions 336
Additional Changes to the Assembler 337
Case Insensitivity 337
Label Formats 337
Quoted String Syntax 337
Assembler Statistics 338
Eight Byte Integers 338
S Attribute 338
Spelling Changes 338
IEEE Directive 339
MSB Directive 339
RENAME Directive 339
SETCOM Directive 340
TRACE Directive 340

Appendix D: Differences Between ORCA/M 4.0 and ORCA/M 4.1 341

The Command Processor and Utilities 341
The Assembler 342

Index 343

Chapter 1: Getting Started

How to Use the Manual

This manual is laid out into four major sections. This section, called the User's Manual, is an introduction to the system. The next two sections, called the ORCA HOST Reference Manual and the ORCA 4.1 Assembler Reference Manual, are designed for easy reference and to give a complete, precise explanation of all of the features of the program. The last section is the Macro Library Reference Manual. It lists all of the macros included with ORCA, and describes the required inputs and action performed.

Please note that this manual does not teach you assembly language, it only teaches you how to use ORCA to enter your assembly language files and turn then into executable programs. If you are a complete beginner, you will also need a good beginner's book on assembly language. Several are available, so visit your local computer or book store and pick one up.

If you are new to ORCA, we suggest that you start here, at the beginning, and carefully read the User's Manual. Work all of the examples, and be sure that you understand the material in each chapter before moving on. ORCA is a big system, and like all sophisticated tools, it will take some time to master. It is also the most powerful assembly language development system ever put on a micro-computer, so your time will be very well spent. If the manual seems a bit large, it is because the program can do so much for you.

If you are just learning assembly language, don't try to understand ORCA all at once. Learning assembly language takes time, and you shouldn't try to master the entire tool at the same time as you learn the basics of assembly language programming. The first four chapters of the User's Manual contain enough information to let you write all of the programs you are likely to find in a beginner's book on assembly language. Save the rest for later, when the concepts have sunk in. After writing a few short programs of your own, the advanced features of ORCA will make a lot more sense.

Booting the System

At the front of the notebook is a separate section that includes details on

setting up your system, information on recent changes, and special notes about the system as a whole. The separate booklet allows us to make changes and additions to the way in which the system is initialized and what drivers are available without updating the entire manual. The section is called Read Me First, and you should do just that.

Having initialized the system, you are now ready to start using it. The rest of this chapter will give a very brief overview of the way commands are entered, how the editor is used, and how to assemble and execute a program. By the end of this chapter, you will have entered and executed a working assembly language program.

Some Basic Commands

If you don't have ORCA booted, do that now. When you are finished, you should see a copyright message, and a # character followed by a cursor. This tells you that you are in the command processor. It works a lot like Applesoft BASIC, and we will use that fact to help you get started quicker.

First, type `CATALOG` (followed by a RETURN - we won't usually mention that, but you should always use the RETURN key after entering a command). What you get is very similar to the same command from ProDOS BASIC. Just like with ProDOS, you can enter the name of a directory, and you will get a catalog of the specified directory. If you type `CAT`, you will get a somewhat strange message, telling you that the file is not available. More on that later - for now, just realize that the abbreviated `CAT` command does not exist on ORCA.

This brings up two questions: do you always have to type `CATALOG` (that's a lot of letters), and exactly what commands are available?

The answer to the first question is an emphatic no. Type `C`, then follow it with a right arrow key. The command processor expands the command out to the full `CATALOG` automatically. What it does is to look into the command table and fill in the missing characters from the first command name that matches all of the characters that you typed. If you really wanted `COPY` instead, you can get it by typing `CO` and a right arrow, or by typing `C`, a right arrow, and a few down arrows. In fact, using the up and down arrow keys, you can step all the way through the commands that are available. That's one answer to the second question: you can find out what commands are available by using the up and down arrow keys after expanding any valid command.

There is an ORCA command that makes all of this much easier. Type `HELP`. What you get is a complete listing of the command table. If you have a two drive system, place the /UTILITY disk in the second drive and type

```
HELP CATALOG
```

Each of the commands in the command table has a help file which gives a brief description of what it is for and what parameters it needs. This is really only useful if you have enough disk space to keep the help files around (they are contained in the prefix /UTILITY/HELP).

Well, that's all we will do with the command processor right now. If you are adventurous, you might like to take a break and explore some of these commands. We'll get back to the command processor in the next chapter, but for now, let's move on and edit a simple file.

A Few Editor Commands

In this section, we will take a look at a small handful of the most basic editor commands. The goal is to learn just enough to be able to type in a simple program.

First, let's realize that the ORCA editor is very different from the line editor used to enter BASIC programs. Like good word processors, the ORCA editor is a full screen editor. It is very fast, can edit about 20K, and includes search and replace, block copy and move capabilities. It also supports keyboard macros if you have an Apple //e or Apple //c.

To get into the editor, type the command `NEW`. This loads the editor with an empty file. Another command, `EDIT`, lets you edit a file that already exists, but so far you haven't typed anything in. What you see on the screen is the first twenty-two lines of the file being edited; since it is a new file, they are all blank. Whatever you type will be entered into the edit buffer. Keep in mind that you do not need to hit RETURN, or anything like that, to enter a line. As soon as a character appears on the screen, it becomes a part of the file you are editing. (Nothing gets written to disk, though, until you tell the editor to save it.)

To enter a simple file, you really only need to know nine commands. Four are pretty obvious, if you have ever used a full screen editor. The four

arrow keys let you move up, down, left and right. Even though the RETURN key does not need to be used to enter a line, it is still a handy key - it moves the cursor to the first column of the next line down, getting you ready to type a new line. Finally, the TAB key will be used a great deal. The tab stops appear at the bottom of the screen as a series of ^ characters. (The TAB key is the same as a CTRL-I, which can be used on older Apples that do not have a TAB key.)

Another important skill is inserting and deleting lines of text. To delete a line, move the cursor so that it is on the line you want to get rid of. Now type the ESC key, a Y, and the ESC key again. The line that the cursor was on has vanished, and all of the lines in the file have moved up one to fill in the space. Now try ESC B ESC. This inserts a blank line, moving all of the lines from the cursor on one line down to make room for the new line.

The last command that we will examine right now is CTRL-Q, which leaves the editor. You get five options here - type an N, and then a legal ProDOS file name for now. The file will be saved under the name you supply. After that, type E to leave the editor. That's enough to let you enter the program in the next section. Chapter 3 covers the editor in more detail.

Entering a Simple Program

Listing One shows a short, simple program which writes a line to the screen. Before typing it in, lets take a look at the source code for the program. The first line is an assembler directive. Assembler directives tell the assembler to do something, and usually do not generate code in the finished program. The first directive, KEEP, tells the assembler to keep the finished program, saving it to the file named TEST. It is very important that no other file called TEST be on the default prefix when this program is assembled; if there is one, it will be deleted. Specifically, don't save the source file using the name TEST.

The next line is also an assembler directive. The START directive marks the start of a program segment called MAIN. It happens that this is the only segment in our short example program. (A segment allows your program to be broken up into subroutines with local labels, the way you might be used to in high level languages like Fortran or C.) The assembly language instructions that follow are all in standard format, and look like they would in any 6502 assembler. Near the end of the program, three more directives appear, two DC directives and an ANOP directive. The DC directives will be discussed in Chapter 4; these two define the characters for a message that the program will write to the screen and a RETURN character to go at the

end of the message. The ANOP directive doesn't do anything, it just gives us a place to put the MSG2 label. Finally, the program winds up with an END directive.

```
          KEEP  TEST
MAIN      START

          LDX   #MSG2-MSG1
          LDY   #0
LB1       LDA   MSG1,Y
          JSR   COUT
          INY
          DEX
          BNE   LB1
          RTS

COUT      JMP   ($36)

MSG1      DC    C'Hello, world.'
          DC    H'0D'
MSG2      ANOP
          END
```

Listing 1

Type in the program, just like you see it in the listing. After you have finished, leave the editor, saving the file as MYPROG. Next, type

```
RUN MYPROG
```

In the next few moments, the system will load in the assembler and assemble your program. If you entered it correctly, there will be no errors. (If there were errors, get into the editor with an `EDIT MYPROG` command and correct them, then repeat the `Í` command.) After successfully assembling the program, the link editor is loaded. It takes the output from the assembler and creates an executable program. Finally, the system executes your program, typing the message to the screen.

You have just finished entering and executing a small program from ORCA. The next three chapters will expand on what happened, and teach you a lot more about how to use the system. Having read those chapters, you will be able to do as much with ORCA as you can with most small assemblers, but you will have just scratched the surface of what ORCA can

do for you. The following four chapters examine some of the powerful features that make assembly language programming on ORCA unlike anything you will ever be able to do on any other microprocessor based assembler.

Chapter 2: The Command Processor

Using Wildcards

In the first chapter, we looked at the command processor and explored the use of the arrow keys to expand partial command names and step through the commands in the command table. In this section, we will expand on our knowledge of the command processor.

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

To get a firm grasp on wildcards, we will use the ENABLE and DISABLE commands. These commands turn the file privilege flags on and off. This is similar to locking and unlocking files from BASIC, but you have more control over the process. First, disable delete privileges for all files on the /ORCA directory. To do this, type

```
DISABLE D =
```

Cataloging /ORCA should show that the D is missing from each of the directory entries. This means that you can no longer delete the files. Now, enable the delete privilege for the monitor file. Since the MONITOR file is the only one that starts with the characters MON, we can do this by typing

```
ENABLE D MON=
```

The wildcard matches all of the characters after MON.

What if you want to specify the last few characters instead of the first few? The wildcard works equally well that way, too. To disable delete privileges for the monitor, we can specify the file as =TOR. It is even possible to use more than one wildcard. You can use =.= to specify all files that contain a period somewhere in the file name. Or, you could try M=.=S to get all files

that start with an M, end in an S, and contain a period in between. As you can see, wildcards can be quite flexible and useful.

To return the /ORCA disk to its original state, use the command

```
ENABLE D ?
```

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

Four minor points about wildcards should be pointed out before we move on. First, not all commands support wildcards every place that a file name is accepted. The ASSEMBLE, LINK and RUN commands don't allow them at all, and RENAME and COPY allow them only in the first file name. Secondly, wildcards are only allowed in the file name portion, and not in the subdirectory part of a full or partial path name. For example, /=/STUFF is not a legal use of a wildcard. The next point is that not all commands respect the prompting of the ? wildcard. CATALOG does not, and new commands added to the system by separate products may not. (Later, we will look at how commands can be added to the system.) Finally, some commands allow wildcards, but will only work on one file. EDIT is a good example. You can use wildcards to specify the file to edit, but only the first file that matches the wildcard file name is used.

Required and Optional Parameters

There are two kinds of parameters used in commands, required and optional. If you leave out an optional parameter, the system takes some default action. For example, if you use the CATALOG command without specifying a path name, the default prefix is cataloged. An example of a required parameter is the file name in the EDIT command: the system really needs to have a file name, since there is no system default. For all required parameters, if you leave it out, the system will prompt for it. This lets you explore commands, or use commands about which your memory is vague, without needing to look them up.

Types of Commands

Although they are all used the same way, there are really three distinct kinds of commands supported by ORCA. The first kind is the built-in

command; that is the kind you are probably most familiar with. The code needed for a built-in command is contained right in the MONITOR, so it is always available. This is the type of command supported by DOS and ProDOS.

A second kind of command is the language. ORCA is set up to support multiple languages. This entire topic will be covered later, when we look at EXEC files in Chapter 5.

The last kind of command is the utility. Utility commands are commands that use a separate program to do their function. PEEK is an example. When you use a utility command, ORCA must find and load a completely separate program, them reload the monitor when the command has finished. If the /UTILITY disk is in a disk drive while this is happening, ORCA will do all of this without bothering you, but if it is not, you will get a message telling you to put the /UTILITY disk into a drive. The reason that utility commands are used at all is to be able to add more commands to the system than can be fit into memory at one time. In addition, you will be able to add your own commands to the system. This is discussed more fully in Chapter 9.

The ORCA Monitor Commands

The table on the next page lists the commands that are available in ORCA. Beside each is a brief description of what it is for. Many of the names may be familiar to those who have used DOS or ProDOS extensively, and will work very much like what you are used to. Rather than go through them one by one, we will let you scan the list and pick out the ones that look useful to you. You can use the built in help facility to find out more, or you can flip back to the reference manual for a complete description.

ASML	Assemble and link.
ASMLG	Assemble, link and go.
ASSEMBLE	Assemble.
CATALOG	Catalog a subdirectory.
CHANGE	Change language of source file.
CMPL	Compile and link.
CMPLG	Compile, link and go.
COMMANDS	Modify command table.
COMPILE	Compile.
COMPRESS	Compress and/or alphabetize directories.
COPY	Copy files.
CREATE	Create new subdirectories.
CRUNCH	Compress and/or alphabetize directories.
DCOPY	Copy complete disks.
DELETE	Delete a file.
DISABLE	Disable file attributes.
DISASM	Disassembler.
EDIT	Edit a source file.
ENABLE	Enable file attributes.
HELP	List the commands in the command table.
INIT	Initialize the disk.
LINK	Link an object module.
PEEK	Disk zap utility.
PREFIX	Change the default prefix.
QUIT	Return to ProDOS.
RENAME	Change a file name.
SCAN	List routines in an object module.
SET	Set system attributes.
SHOW	Show system attributes.
SWITCH	Change the order of a file in a directory.
TYPE	Type a source file.
XREF	Generate a cross-reference.

Chapter 3: The Editor

Introduction

What follows is a guided tour through the most useful text edit commands. Not all commands are covered here, and not all capabilities of the commands listed are dealt with. Enough information is given to learn how to enter normal assembly language programs.

We will need a file to practice on, so start by placing the /MACROS disk on line. Now enter the text editor by typing

```
EDIT /MACROS/M6502.MSC.
```

Moving Through the File

The display will clear and the first twenty-two lines of the file will appear on the screen. We have already seen that the cursor keys can be used to move through a file. Another way is to enter the ESC mode. The editor runs in two modes: edit and escape. The escape mode is simply a way of making more commands available than there are keys on the keyboard. It works very much like the I-J-K-M escape key commands in the Autostart ROM.

Press the ESC key. At the bottom of the screen, the message

```
ESC
```

appears to indicate that the escape mode has been entered. This message stays there for as long as the escape mode is active, and vanishes upon return to the edit mode. Leave the escape mode now by keying ESC again. Any key that is not an escape mode command key (for example, ESC) causes the escape mode to be exited.

Re-enter the escape mode (by using ESC again) and type `X`. A RETURN is not needed. The screen will change to show the next twenty-two lines of text. Now type a `W`. This moves the display back to the first twenty-two lines of text. Notice where the keys are on the keyboard. Think of the screen as a window which is laid on a giant scroll; it makes sense to use a

top row key like W to move toward the beginning of the file and X to move toward the end. Try these commands a few times to get a feel for them.

Just to the right of the W and X keys are the E and C keys. These move the text window one line, instead of one page. Try them.

At this point, the screen can be moved up and down in the file. Cursor movement is next. Just as with the autostart ROM, the I, J, K and M keys move the cursor. Note that these keys are placed on the keyboard in a diamond, their positions indicating the direction each key moves the cursor in. Try them out. Also try and move the cursor off of the screen in each direction, and see what happens. When using the keys, notice that J will not move the cursor to the left of the first column. K is similarly limited by the end-of-line marker in the TAB line, which is set to column fifty-nine. I cannot move the cursor beyond the top of the first line in the file. However, M will continue to move past the end of the file, adding blank lines to the end of the file.

Having moved to the end of the file using the M key, it would be nice to have a quick way to get back to the start of the file. One way to do this is to use the repeat feature. While in the ESC mode, press the 9 key. Nothing happens. Now press W. Instead of moving up one page, the screen scrolls up nine pages. Numbers up to 255 may be entered in this manner. This feature will work with any ESC command; in fact, the grouping of functions as either escape commands or control key commands was largely determined by selecting those commands which could most benefit from having the repeat feature available to them.

At this point, you should be somewhat adept at moving through the file in memory. There are a few other movement commands that are very useful, but they are used from the edit mode. Get back into the edit mode (using ESC) and try CTRL F and CTRL L. These keys are used to jump to the beginning or end of a file in one step. Notice that these keys are not located geographically; instead, they were assigned mnemonically. CTRL F stands for first line, while CTRL L stands for last line. Now try CTRL T and CTRL B. These commands move to the top and bottom of the display window. Finally, CTRL E moves to the end of a line, and CTRL W, located to the left of E, moves to the beginning of a line.

The next feature for moving the cursor is the tab function. The tab keys are TAB (or CTRL I) and CTRL S. TAB moves the cursor forward to the next column marked by a ^ character in the line below the screen. CTRL S moves you back one tab stop.

Entering Text

Text may be entered anytime the edit mode is active, i.e., when you are not in the escape mode. Move the cursor to where the text is to be, and start typing. The characters typed replace the old contents of the file, and the cursor moves one space to the right.

Inserting and Deleting Text

As changes are made to a file, one of the more frequently needed tasks is inserting blank lines and removing unwanted ones. As we saw earlier, this is done from the escape mode.

As with the cursor movement commands, the insert and delete commands are geographically oriented. The diamond of keys immediately to the left of I, J, K and M are Y, G, H and B.

To delete a line, begin by placing the cursor on it. It doesn't matter where in the line the cursor is located. Press `Y` from the escape mode. The line vanishes, and the following lines are moved up one line to fill in the space. The repeat feature also works; to delete twelve lines, type `12Y`. The twelve lines starting with the line that the cursor is on are deleted. The lines do not have to be on the screen to be deleted.

Inserting blank lines is similar. When B is hit in the escape mode, a blank line is inserted at the location of the cursor. The line that the cursor was on and all of the lines after it are moved down to make room for the new line. Again, the repeat feature works.

To insert characters, place the cursor at the place that a new character is desired. The H key will cause all characters from the cursor position on to be shifted to the right one character. A blank is placed at the cursor position. Try this enough times to cause a character to go off of the end of the line. Now use the G key to delete one of the blanks just inserted. A blank appears in the last column. This emphasizes that characters scrolled off of the end of the line are lost.

One of the more common uses of the line insert capability is to make room for large additions to source files. The cursor is placed appropriately, then a few hundred lines are inserted followed by a stream of text. There will undoubtedly be some extra blank lines left over. Typing `CTRL R` from the

edit mode removes these lines, beginning at the cursor position and continuing to the first non-blank line.

To see how this works, insert 50 blank lines. Now move the cursor down a couple of lines, and enter the edit mode. Type CTRL R.

Moving and Copying Text

The editor lets you copy or delete sections of a file to disk, and read existing disk files into the one being edited. Selecting the lines to write out is a two step process: first, move the cursor to the first line to be copied or deleted, and press CTRL O. The message MARK appears at the bottom of the screen, letting you know that you have marked one end of a block of text. Now move the cursor to the last line in the block and type CTRL O again.

At this point, you are asked for a file name. This is the name of the file to write the lines to. If you simply hit RETURN, the editor writes the lines to a special file (called SYSTEMP) in the work prefix. Next you are asked if the lines should be deleted. Answer with Y or N.

After doing this, move the cursor to another point in the file and type CTRL P, for pop lines from buffer. Again, you are asked for a file name. You can enter the name of any ProDOS TXT or ORCA SRC file, or simply hit RETURN. The lines from the file are inserted at the cursor position.

Single characters can also be removed from the file and placed in another buffer, the character buffer. Unlike the copy buffer, the character buffer functions like a true stack. Characters are deleted using the DELETE key, and replaced using the open apple and DELETE key together.

As characters are deleted from the file they are placed in the character buffer. As they are retrieved, they are removed from the buffer. Placing more than 256 characters in the buffer starts to delete the oldest characters. Retrieving more characters than were put in places blanks on the screen.

Search and Replace

The last feature of the editor to be examined here is one of the most useful and powerful. The editor is capable of searching for strings in the source file and replacing them with a different string.

Enter the escape mode and type an asterisk (*). An asterisk is shown at the bottom of the screen, followed by the cursor. Type `END`, followed by a RETURN. Notice this causes a return to the edit mode, rather than remaining in the escape mode.

A search string has just been entered. It remains valid until a different search string is entered. Typing a CTRL X will move the text window so that the next occurrence of the search string in the file appears on the top line, with the cursor placed at the start of the string. CTRL Z does the same thing, but searches toward the beginning of the file instead of toward the end. Try these commands, paying special attention to what happens when the string is not found.

Replacing the search string with a replace string is almost as easy. From the escape mode, type a colon (:). Again, the prompt appears at the bottom of the screen. Type `FIN` and a RETURN. This enters a replace string. CTRL V initiates a search and replace starting at the current cursor location and proceeding down in the file, while CTRL C does the same going up in the file. Try one of these commands.

After entering either command, the editor asks if an automatic or manual search and replace should be done. An automatic search and replace replaces every occurrence of the search string with the replace string. For now, try a manual search. After entering the M, if the search string is found, the line containing it is placed at the top of the screen. At the bottom is the question

```
Replace (Y N Q)?
```

Any reply starting with Y replaces this occurrence of the search string with the replace string. N will skip to the next occurrence of the search string without changing this one. Q quits the search and replace sequence.

Experiment with these commands for a while. Pay close attention to results when the search string cannot be found. Also, be sure and try search and replace commands with search and replace strings of different lengths, and with blanks at the beginning and end of the strings.

Other Features

The editor has some other commands, including the ability to enter all ASCII characters from the Apple][+ keyboard, keyboard macros, and an insert mode. These advanced features are left to the reference manual.

Chapter 4: The Fundamental Assembler Directives

This chapter is a guided tour through the most useful assembler directives. ORCA/M has a large complement of assembler directives, easily the most complete set of any microcomputer based assembler. Despite this fact, there are only a few which are absolutely necessary to write assembly language programs. In this chapter, we will take a close look at the eleven most important assembler directives, as well as how lines are commented and some of the rules for coding operands. Once these directives are mastered, you will be able to use ORCA to assemble assembly language programs that appear in magazines and beginners books on assembly language. The directives that we will look at are:

ANOP	Assembler no-op
APPEND	Concatenate two source files
DATA	Start a new data segment
DC	Declare constant bytes
DS	Declare storage
END	End a code or data segment
ENTRY	Declare global label
EQU	Declare a constant
GEQU	Global equate
KEEP	Keep output
START	Start a new code segment

The Assembly Language Statement

There are two kinds of lines in an assembly language program, comments and statements. The assembly language statement will be covered here; comments will be covered in the next section.

Assembly language statements can be broken down into three groups. The first is the assembly language instruction. Each assembly language instruction corresponds exactly to a single machine language instruction that the computer can understand. For example, the assembly language instruction RTS maps to the matching language instruction $60. The way that these instructions are coded is very standard, with almost every 6502 assembler using exactly the same format. It is that fact that makes it

possible to use ORCA to assemble a program written with another assembler, with just a few minor changes to the directives used.

The second kind of line is the assembler directive. Assembler directives look a lot like assembly language instructions, but in fact they are very different. While an instruction corresponds to a machine language instruction, and tells the computer to take some action in the finished program, the directive does not. A directive tells the assembler itself to do something. An example of this would be the KEEP directive which instructs the assembler to keep the object module created under the name provided in the operand field. There is very little standardization in directives. Each assembler is just a little different from the others. The directives in ORCA were patterned after the assembler that we believe to be the most powerful ever written, the IBM 370 assembler.

Finally, there is the macro. Macros are expanded by the assembler to produce one or more instructions or directives, letting you do very complex things with very simple statements. The ADD2 macro, supplied with ORCA, will perform the function of the seven assembly language instructions normally needed to do this operation in one line. Chapters 7 and 8 explore macros in detail.

Each assembly language statement, whether it is a macro, directive or instruction, has four distinct parts. One of them, the operation code, or op code, is required on every statement in ORCA. The op code is the name of the statement, like JSR for the jump-to-subroutine instruction, or KEEP for the keep directive. Unless there is a label, the op code can start in any column from two to forty. It is customary to place it in column ten, so the editor has a tab stop in that column.

Most statements can have a label, and in fact, a few directives actually require one. Labels serve the same purpose as line numbers in BASIC, giving you a way of telling the assembler what line you want to branch to or change. Labels must start in column one. They must start with an alphabetic character, tilde or underscore. The remaining characters can be alphabetic (A through Z), numeric (0 through 9) or the underscore character or tilde. Only the first ten characters are significant, although you can use as many as you like. Thus, VERYLONGLABEL and VERYLONGLABEL2 refer to the same label. The underscore is significant, so MYLAB and MY_LAB are not the same. If a label is used, there must be at least one space between it and the op code. Finally, ORCA is case insensitive. This means that you can use a lowercase letter anywhere that an uppercase letter is used, but they mean the same thing.

The third field in an assembly language statement is the operand field. There must be at least one space between the op code and the operand. It can start in any column before column forty, and customarily starts in column sixteen, where the editor has a tab stop. Operands vary a great deal. Instruction operands are explained in books about the 6502. Operands for directives and macros are described as the directive or macro is introduced. However, one issue is important in all types of statements. You can substitute mathematical expressions for any number, including multiplication, division, addition, subtraction, bit shift operations, etc. These expressions are written the same way that you would write them in most high level languages. Since it is so natural, we won't discuss it in detail here. See page 128 if you would like to know more. The examples below illustrate the main points.

```
            LDA   LABEL+1
            DS    LENGTH*400
            DS    (LENGTH+1)*400
L1          EQU   L0/14+1
N1          EQU   -1
```

The last field in an assembly language statement is the comment field. There must be at least one space between the operand (or op code, if there is no operand) and the comment. The comment is for your benefit. It does not effect the finished program in any way, but it does help you to remember what the program is doing. Some assemblers require semi-colons before the comment; ORCA does not. Comments normally start in column forty-one. As you would expect by now, the editor has a tab stop there.

Comment Lines

The second type of line that can appear in an assembly language program is the comment. Comments are used to help you remember what a program is doing. Comments do not effect the finished program in any way. Specifically, unlike BASIC, comments do not take up room in the finished program, so there is no reason to avoid them.

ORCA supports five kinds of comment line. First, a completely blank line is treated as a comment. Any line with an *, ; or ! in the first column is also treated as a comment, and any keyboard character can be used after the first character. Finally, a line with a . in column one is a special kind of comment line called a sequence symbol. Sequence symbols are not printed

in the listing produced by the assembler. They are used by conditional assembly directives, and discussed in detail in Chapter 8. For now, if you decide to use this form of comment to get a line that shows up in the editor, but not later in the listing, be sure and place a space after the . character.

KEEP

We saw the KEEP directive in the short program that we wrote back in Chapter 1. The KEEP directive tells the assembler to keep the object modules that it produces. The operand field can contain any valid ProDOS path name. The current prefix is used if only a partial path name is specified. Only one KEEP directive can be used in a program, and it must appear before the first START directive. Although it is possible to create a finished program without using the KEEP directive (see the description of ASSEMBLE in the reference manual to find out how), most programs do in fact start with a KEEP directive.

START and END

The START and END directives are much more powerful than you might expect from looking at our first program back in Chapter 1. These directives are used to indicate the start and end of named code segments. The END directive has no operand, and usually no label. The START directive also has no operand, but requires a label. The label on the START directive becomes the name of the code segment. A program can, and usually does have more than one code segment. It is not necessary to do separate assemblies to assemble each of the code segments, nor is it necessary to put them in separate disk files.

Inside a code segment, all labels that are not defined using the GEQU or ENTRY directives are local labels. (The GEQU and ENTRY directives are covered later.) This means that no other code segment can see the label. For example, the following program would produce an error in SEG1 because LAB1 is not defined in SEG1, but it is perfectly legal for both segments to use LAB2.

```
SEG1    START
LAB2    LDA   LAB1
        END


SEG2    START
LAB2    LDA   LAB2
LAB1    LDA   LAB1
        END
```

The concept of local labels is very powerful, and unfortunately, very rare in assemblers. Because a section of code can be developed independently of all other code in the program, you can build up a library of subroutines that can be moved from one program to another. And, unlike other assemblers, you don't have to worry about whether you have used the label LOOP somewhere else in the program. Its perfectly all right to have the label LOOP in every segment in the program, so long as it is used only once in each segment.

As you start to write long programs with ORCA, you should use the idea of the segment by dividing your program into short subroutines. As with the more advanced high level languages, these subroutines can be developed and debugged separately. Used properly, the program segmentation provided by the START and END directive can be one of the most powerful aids to writing large assembly language programs.

Equates

One of the basic ideas behind structured programming is to give meaningful names to numbers. This is done by defining a constant, which is used instead of the number. In assembly language, constants are defined using the EQU directive, and are called equates.

To define a constant, place the EQU in the op code field, the name of the constant in the label field, and the value in the operand field. For example,

```
ONE     EQU   1
TWO     EQU   1+1
FOUR    EQU   TWO*TWO
```

As shown, you can use expressions in the operand, and you can use constants defined by earlier equates. Keep in mind that equates are used to define constants: each term in the expression in the operand field must have

a specific value when the EQU is encountered. One of the most common problems that result from this fact is people trying to use the construct

```
HERE     EQU   *                      Incorrect
```

to set the label HERE to the address of the current location counter. Because ORCA uses a link editor, the value of the current location counter is not known at assembly time, and the operand is not a constant. If you need to define a label without generating code, you can use the ANOP (assembler no-op) directive:

```
HERE     ANOP                         Correct
```

Constants should be defined before they are used. It is customary to put all of the equates in a segment right after the START directive. This is only a strict requirement when the constant is used later as a zero page address (or as a long address on the 65816), but it is better to get in the habit of defining before use to avoid problems later.

DC and DS

A program consists of instructions and data. So far, we haven't found out how to put data into our assembly language programs. In ORCA, this is done with two directives: DC, or define constant; and DS, or define storage.

The DC directive is used whenever you want to put an initialized value into memory. With the DC directive, you can put characters, binary or hexadecimal values, integers (in a variety of lengths), addresses, or floating point numbers into memory. In this section, we will only talk about characters, integers, and hexadecimal values. If you need to enter one of the other kind, see Chapter 16.

The operand of the DC directive tells what values will be placed in memory in the finished program. The first letter is a format specifier, which tells what format the information is in. The table below lists all of the valid format identifiers.

A	address
B	binary
C	character
D	double precision floating point
F	single precision floating point
H	hexadecimal
I	integer
R	hard reference
S	soft reference

Table 2: DC Format Specifiers

Let's start by looking at a declaration of a string of characters. The DC statement shown below defines the string that is enclosed in quote marks, and illustrates that quote marks inside the string must be doubled.

```
DC C'Now''s the time for all good people to use ORCA.'
```

The format shown here is very similar for all types of DC directives. The op code is always DC, the first character is the type of data to be defined, and the data follows, enclosed in quote marks. When you are defining data where several different values of the same type are coded, you can separate the individual values with commas, as seen in the following example for integer declarations. Note that integers are always stored least significant byte first, which is the way the 6502 likes its addresses, and also the way the ORCA math libraries like to find numbers.

```
DC    I'1,1+1,3,4'
DC    I8'1000000000000'
DC    50I'1'
```

The second example demonstrates that integers come in several lengths. If you use an I for the type specifier, you will get a two byte integer. However, the I can be followed by any number from one to eight, giving an integer with that many bytes. These large integers can represent very large numbers; the eight byte integer shown on the second line can represent numbers from -9223372036854775808 to 9223372036854775807. Finally, the last line introduces the idea of a repeat count, which can be used with any type of values that are placed one after the other in memory. The last statement, then, would initialize 100 bytes of memory. There is no limit to the number of bytes that a single DC directive can define, but the repeat count is limited to 255.

The last type of DC directive that we will look at here is the hexadecimal DC definition. Hexadecimal digits include the numbers 0 to 9 and the hexadecimal digits A to F. Again, since the assembler is case insensitive, you could use lowercase letters if you like. The only thing that can appear between the quote marks is hexadecimal digits and spaces. Each byte of memory can contain two hexadecimal digits, and the digits coded in the DC directive are placed in memory in pairs. If you code an odd number of digits, the last nibble of the last byte is padded with a zero. The two DC directives in the example below produce exactly the same thing: Two bytes, the first of which contains a $B1 and the second of which has a $D0.

```
            DC    H'B1 D0'
            DC    H'B 1 D'
```

The last example shows how you can mix data types on a single line. For a very reasonable example, lets assume that you want to place a carriage return code at the end of a line of characters. Since the carriage return code is a $0D, we could code it in hexadecimal, as in the first example. Or, if you prefer, we could define a symbol called RETURN, and code it as a one byte integer, as in the second DC directive.

```
            DC    C'Error!',H'0D'

RETURN      EQU   $D
            DC    C'Error!',I1'RETURN'
```

As mentioned earlier, there are several more data types supported for special uses. For more information, see the reference manual, starting on page 136. Also, note that the DC directive and character constants used in expressions default to standard ASCII, with the most significant bit cleared. This is correct for most peripherals and for the subroutine libraries, but if you are writing your own subroutines to output directly to the Apple screen, you may want to have the most significant bit set. The MSB directive, described on page 146 of the reference manual, gives you a way to do that.

Before moving on, lets take a quick look at another way to define data. The DS directive is normally used when you don't care what initial value a variable has, but you want to reserve some space in the finished program. The operand of the DS directive is a constant which tells how many bytes to declare. The bytes are reserved in memory and initialized to zero.

```
NUM       EQU   10
SPACE     DS    100    DEFINE 100 BYTES OF FREE SPACE
          DS    NUM*2  DEFINE NUM TWO BYTE INTEGERS
```

Global Labels

So far, we have only talked about local labels, where a label in one code segment is "invisible" in all others. There are, of course, times when you want another segment to be able to see a label. We already saw one way of doing this: the label on the START directive is global. This means that every segment in the program is able to see the label defined on a START directive. By the way, it's all right to define a local label with the same name as a global one, so long as they are not defined in the same segment. The assembler will choose the local label in preference to the global one.

There are also two other directives which can define a global label. The first is a global form of the EQU directive, used for defining global constants. It works just like EQU, but the op code is GEQU. You should realize one important difference between the labels defined with the GEQU directive and all other labels. Those defined with GEQU can be seen at assembly time, while all the other global labels can only be seen at link time. This means that you should always use a GEQU directive to define a zero page (or long address on the 65816) label that will be used in more than one subroutine. That way, the assembler can automatically decide which addressing mode is appropriate.

The ENTRY directive can also be used to define a global label. The ENTRY directive has no operand, and its label receives the value of the current location counter. Its most common use is to define an alternate entry point into a subroutine, as shown in the example below.

```
SUB1     START                PRINT A HEX BYTE
         LDA   #$AB
         JSR   SUB2
         JSR   SUB2A
         RTS
         END

SUB2     START                PRINT MSB OF HEX BYTE
PRHEX    EQU   $FDE3
         PHA
         LSR   A
         LSR   A
         LSR   A
         LSR   A
         JMP   LB1

SUB2A    ENTRY                PRINT LSB OF HEX BYTE
         PHA
         AND   #$F
LB1      JSR   PRHEX
         PLA
         RTS
         END
```

DATA Areas

Especially in large programs, it is nice to be able to build data areas where global variables and constants can be stored. In fact, it is even nice to be able to have several such areas, very much like the way Fortran allows common areas to be defined. ORCA has a structure like this, called the data area.

Data areas are in fact separate segments. But instead of using a START directive, data segments start with a DATA directive. They still end with an END directive. Only data definitions are allowed inside of a DATA area; the assembler will flag an instruction as an error. Labels in a data area are still local labels, and normally can only be seen inside of the data area. Another directive, the USING directive, is then used in any code segment that needs to use the labels in the data area. The USING directive has the effect of making the labels in the data area local to both the data area and the segment where the USING directive appears. More than one code segment can use the same data segment, and in fact, more than one data

segment (up to 127) can be defined and used, in any combination, by a code segment. The syntax for these two new directives is illustrated below.

```
MAIN      START
          JSR   INIT
          JSR   PRINT
          RTS
          END

COMMON    DATA
KEYBOARD  EQU   $C000
PRBYTE    EQU   $FDDA
NUM1      DC    H'CD'
NUM2      DS    1
NUM3      DS    1
          END

INIT      START
          USING COMMON
          LDA   KEYBOARD
          STA   NUM2
          RTS
          END

PRINT     START
          USING COMMON
          CLC
          LDA   NUM1
          ADC   NUM2
          STA   NUM3
          JSR   PRBYTE
          RTS
          END
```

The APPEND Directive

As you write larger and larger programs, you will eventually run out of space in the editor. When this happens, you can split your program into two sections and use the APPEND directive to tell the assembler that another file belongs in the finished program. The operand of the APPEND directive is a ProDOS path name. It works like a GOTO, in that any line after the APPEND directive is ignored. The directive can be used to append a file

that is on a disk which isn't even in the computer when the assembly starts. When the APPEND directive is encountered, the assembler checks for the disk. When it is not found, it stops and asks you to place the disk in a drive, reminding you of the name of the disk it wants. If you have made an error, and there is no such disk, just hit the ESC key. Effectively, this means that you can assemble any program on a one drive system that you could assemble on a two, three, or more drive system, although swapping disks will be necessary.

Chapter 5: Advanced Commands

With the basics behind us, we can start looking at some of the advanced features of ORCA. You should keep in mind that even in these chapters we will not look at all of the capabilities of ORCA. The real purpose here is to cover the most commonly used features, as well as those techniques that require the use of features described in different parts of the reference manual. After getting comfortable with the material in these four chapters, you should plan on skimming the reference manual to look for topics which interest you.

Linking to Several Locations

Let's start by looking at one part of the ORCA system that we have ignored up to now. The link editor can, as we have already proved, be ignored. On the other hand, it can also be used to do some very powerful things. The first of those that we will look at is assembling a program one time, then linking it several times to get binary files that will load to different locations. The most common reason for doing this is when we are writing something to work with BASIC. Frequently, you will want several different versions of the assembly language parts that load at different places.

To do all of this, start out by simply doing an assembly. The ASSEMBLE command works just like the RUN command that we have used up to this point, but instead of doing an assemble-link-execute sequence, it only does the assemble. The result of the assembly is the OBJ files that you find when you do a catalog of the disk. These files cannot be executed yet. They have been assembled into a form that contains all of the information necessary to generate code for any location in memory. This is what is known as a relocatable object module, or simply object module.

Now we can start generating the binary modules for different locations. The binary modules are the BIN files, and they are the ones that can actually execute. Each of the BIN files will only work if it is loaded at the proper place in memory. If you check the reference manual, you will find that there is a parameter called ORG on the link command. The ORG is the origin for the final binary file, and by specifying it, we will get a binary file that will execute at the point we tell it to. For example, if the output file from the assembly was called MYPROG, it would produce the files MYPROG.ROOT and, assuming there is more than one code segment,

MYPROG.A. Then to get two versions of the BIN file MYPROG, one of which executes at $4000 and one of which works at $300, we would type

```
LINK MYPROG KEEP=MYPROG.300 ORG=$300
LINK MYPROG KEEP=MYPROG.4000 ORG=$4000
```

The KEEP parameter tells what file name to use to save the binary file. It isn't required, but the binary file isn't produced if you don't use it. As you can see, we can use hexadecimal numbers in the ORG parameter. We could also use decimal numbers, in which case the $ character is omitted.

Two final points before we move on. First, you can do a perfectly good link edit by leaving the ORG parameter off. In that case, as with all programs that do not specify an ORG explicitly, the program will start at $2000. This is the best place to start a program that will run under ORCA, unless you are using the high resolution graphics page, in which case you should ORG the program to $4000, or, if page 2 will be used, $6000. You can run the program under ProDOS that starts at $800, but such a program cannot be executed under ORCA, since the ORCA operating system extension uses the memory from $800 to $1FFF.

Secondly, if you are going to specify ORG values in the link edit step, you must not use ORGs inside the program. An ORG in the program will override the ORG you specify in the link edit step. If you would like to use an ORG directive in the program, you can read up it on page 147 of the reference manual.

By the way, you can do an assemble-link sequence without executing the finished program using the ASML command. Its syntax is the same as the RUN command.

Partial Assemblies

The most common use of the link editor under ORCA is one that doesn't even require the use of the LINK command: we can still use the old RUN command. It is also a feature that is not provided on any other system that we know about. This is the idea of the partial assembly.

Lets imagine for a moment that you are no longer a beginner with the ORCA system. Your program has grown to about 3000 lines of code, and contains about 40 code segments. Right in the middle, there is a single line of code that has to be changed, in the code segment called BUG. With other assemblers, you would change the line and reassemble the entire

program. With ORCA, you would do a partial assembly instead. The command would look like

```
RUN MYSOURCE NAMES=(BUG)
```

What happens is this: the assembler recognizes that you are doing a partial assembly from the fact that you specified a NAMES parameter in the command line. It scans your program, resolving directives like APPEND and GEQU which effect all of the code segments, but skips lines that effect code segments that you have not asked for. When it finds BUG, it assembles that subroutine, then quits. Assuming this is the first partial assembly that you have done, BUG is placed in an OBJ file that ends in .B instead of the usual .A. The link editor, when it starts up, will start with the file ending with .ROOT (which contains the first code segment in your program), and continue with the file with the highest alphabetic suffix, in this case, .B. After all of the subroutines in the .B file are linked, the linker moves back to the previous file, in this case ending with .A. It skips any segments that it has already found, so the old version of BUG is not included in your program.

More than one segment can be assembled on a partial assembly. To specify multiple segments, enclose all of the names in the parentheses and separate them with commas or spaces. You can do up to twenty-five partial assemblies. At that point, the suffix is .Z, and the next suffix produces an illegal file name.

Two difficulties can occur when doing partial assemblies, both of which are solved using the CRUNCH utility. The first, and most obvious, is that you will use a lot of disk space doing all of those partial assemblies. The CRUNCH utility takes the files produced by a partial assembly and combines them all into a single .A file again, deleting duplicates of the subroutines. The other problem arises when the order of code segments is critical. The idea of the code segment is that it is a separate section of the program whose order in relation to other segments is not important. However, there are cases when the order is, in fact, very important. When that happens, the fact that the CRUNCH utility restores the original order of the object modules can be useful. If you are concerned about the order, a partial assembly and link edit becomes a three step process. If you are assembling a program called SOURCE which produces a final file called OBJECT, the sequence would look like this:

```
ASSEMBLE SOURCE NAMES=(SUB5 SUB15)
CRUNCH OBJECT
LINK OBJECT KEEP=OBJECT
```

There are three cases when you must bite the bullet and do a full assembly. They are:

1. When a segment is deleted or renamed.
2. When a global equate is changed.
3. When order is important and a segment has been added.

If you would like to read more about how the link editor works, refer to Chapter 13. The CRUNCH utility is described on page 77, and a utility that lets you see what segments are in an object module, called SCAN, is described on page 83.

Subroutine Libraries

After linking your program, if there are any labels used that have not been found, the link editor automatically scans the SUBLIB prefix for libraries. Any object module found there is treated as a library. It is scanned, and if the name of the segment matches the name of one of the missing labels, the segment is included in your program. Each subroutine library is searched exactly one time, and is searched sequentially. Segments in the library can refer to other segments, provided the other segment comes after them in the library.

To create your own library, write the subroutines as a separate program, then include a short dummy subroutine at the top of the program. Do an assembly. (Link editing is not required.) Now, delete the .ROOT file, which contains the dummy segment from the top of the program, and copy the library to the SUBLIB directory. If the order that the libraries will be searched is important, use the SWITCH command to change the order of the files in the SUBLIB prefix; they will be searched in the order listed by the CATALOG command. The SWITCH command is described on page 85.

If you are using a library and need to include a segment which, for some reason, would not normally be included, you should use the R type DC directive. That directive causes a hard reference to the segment. This tells the linker to include the segment (or generate an error if it cannot be found) even if the linker doesn't think the segment is needed. The directive does

not use up any room in the final program. For example, to force SUB1 and SUB2 to be included from a library, you would use

```
        DC     R'SUB1,SUB2'
```

EXEC Files

ORCA supports a command file that works a lot like EXEC files under BASIC. In the command file, you can enter any sequence of commands that you can type from the keyboard, then run the EXEC file. Like executing a binary file, you just type the name of the EXEC file to execute it.

There is one catch to using EXEC files; they must be stamped as EXEC files. If you catalog a directory that has some ORCA source files on it, you will notice that they all say ASM6502 in the auxiliary type field (the last entry on the line). This tells you that the system thinks that the file is supposed to be sent to the assembler. An EXEC file must say EXEC in that field, so that the system is sure that the file is to be used as an EXEC file. To do this, you must change languages before creating the new file with the editor. To change to the EXEC language, simply type EXEC. Now, if you enter the editor with the NEW command and create a new file, it will have EXEC in the auxiliary field. Be sure and switch back to ASM6502 by typing that language name before creating new assembly language files!

There is another way to change the language that will be used when you edit a new file, and that is to edit an existing file. The current language will change to match the language of the file you have edited. This brings up two problems: how to tell just what language you are creating, and how to change the language of an existing file. To see what the default language is, simply type

```
SHOW LANGUAGE
```

The system will print out the language that will be used on a new file. (Incidentally, the SHOW command can also list the available languages, tell you what time it is, and list the volume names of the disks that you have in each disk drive. See page 85 of the reference manual, or type HELP SHOW.)

To change the language on an existing file, you can use the CHANGE command. Two parameters are required. The first is the name of the file to change, which can of course be specified with a wildcard if you are

changing several files at once. The second parameter is the language to change the file to. So, to change COMFILE from whatever it is to an EXEC file, you can type

```
CHANGE COMFILE EXEC
```

Please take a moment to experiment with these ideas. Getting used to them now can save you a lot of time and difficulty later.

By the way, there is one very special EXEC file that you should know about. When the ORCA system boots, it looks on the SYSTEM prefix for an EXEC file called LOGIN. If one is found, it is executed as part of the boot up sequence. This lets you set default prefixes, show the log in time, or even run a program right away. (A calendar program would seem appropriate.)

Note that EXEC files cannot contain other EXEC files.

Device Numbers

If you have been experimenting, you may have noticed that ORCA does not support slot and drive parameters. Instead, you can use device numbers. This capability is copied from the Apple /// SOS operating system.

The idea is that every disk has a device number, which ranges from one to the number of disk drives that you have. You can specify the device by substituting a .D and the number of the drive instead of / volume name in a full path name. Thus, to catalog the STUFF directory on the disk in slot 6, drive 1, you would type

```
CATALOG .D1/STUFF
```

Device numbers are assigned in volume search order. To see what the numbers are, as well as what volumes you have in each device, type

```
SHOW UNITS
```

Redirecting Input and Output

As you may know, the Apple has always used two hooks in zero page to redirect input and output. As a result, most programs written for the Apple send output through a hook at $36, and receive input through $38. These

hooks are supported even if you aren't using them on purpose, since the monitor's standard input and output routines use the hooks. ORCA extends this idea to give you UNIX style I/0 redirection.

Input redirection means that any program that receives input from the keyboard can be told to get it from a disk file instead. Output redirection lets you send output that would normally go to the screen to a disk file or printer. For example, to send a listing of the files in a directory to the printer, you could type

```
CATALOG >.PRINTER
```

The same idea can be used to send the listing from an assembly or link edit to the printer. This capability may seem almost frivolous right now, but after you use it for a couple of weeks, you would be a very unusual person if you would just as soon give it up. Keep it in mind and try using it, and you will quickly find many uses for this ability.

By the way, I/0 redirection can be used inside of EXEC files. Also, the position of the input and output redirections in the command line is not important, since the operating system removes them before the command processor parses the line. You can specify both input and output redirection on a single line, and again, the order is unimportant.

Chapter 6: Advanced Assembler Directives

We have already looked at the assembler directives that are really necessary to assemble a program under ORCA, but of course there are a lot more. The remaining directives can be divided into two broad categories: those that are primarily used for writing macros, and those that generally have nothing to do with macros. In this chapter, we will look at the directives which are not used in macros. Chapter 8 covers the rest.

COPY

The copy directive functions a great deal like the APPEND directive, and like that directive it uses a ProDOS path name in the operand. The only difference between the two is that when you reach the end of the file that was copied, the assembler returns to the original file and continues assembling from the line after the COPY directive, while, with the APPEND directive, the assembler never comes back.

The APPEND directive is more efficient in terms of both time and memory, and it should always be used when you have a choice. The COPY directive is used primarily for those cases where you would like to include a short section of code, for example a series of equates, in several subroutines, or where you have a standard sequence of code that you put in every program.

You can put as many COPY directives in a file as you like. Copied files can append other files, and they can also copy other files. The number of levels that you can copy varies according to how much memory is available, but is generally at least four. This means that file A can copy file B, which copies file C, which copies file D, but if file D also copies a file, you may run out of memory. Then again, you may not. It depends on what other features of the assembler you are using at the time.

Format Control Directives

In this section we will look at several directives that help you format the assembler listing to suit your preferences.

Let's start with the LIST directive, which can be used to turn the listing off entirely. Like many of the format control directives, the operand for the list directive is either ON or OFF. Why would you ever turn the listing off? Well, to start with, errors are printed whether you are listing the file or not, so you can always see them. Add to that the fact that the assembler spends over one tenth of its time writing the listing on the screen for you, and you can start to see why you might not want to list the output.

Now add to this another directive, SYMBOL. It, too, uses ON and OFF as the operand. If you have initialized your system so that a keypress will cause the system to stop when it encounters a carriage return, you will find that the assembler disables this feature - unless you are listing a line of source code or printing a symbol table. If both LIST OFF and SYMBOL OFF have been used, this means that the only kind of line that can cause the system to stop is an error. So you can start a long assembly, hit a key, and leave. If an error is found, the system will wait for you to come back and find it.

Therefore, most programs should start with the following three directives.

```
LIST   OFF
SYMBOL OFF
KEEP   MYPROG
```

By the way, you can also control the speed of the listing with your game paddle or joystick, and can use the ESC key to cause the assembler to abort the assembly. (You must stop the listing first by pressing a key, then use the ESC key.) You can probably figure it out with a little trial and error, or read the details starting on page 120.

Another directive which uses the ON or OFF operand is the ERR directive. If LIST is on, it has no effect, but if list is off, it can be used to turn off listing of error lines.

You may have noticed that the assembler writes the code that it is generating at the left edge of the paper. You may also have noticed that only four bytes are written there, even if you coded a DC directive that generated more than four bytes of code. You can cause the assembler to print all of the code in a DC directive (up to a maximum of 16 bytes) by using the EXPAND directive, which, like the other directives, takes an operand of ON or OFF. Keep in mind, though, that the assembler will still

need a line for each four bytes that it shows you, so you might not want to use this directive!

Two directives let you control output sent to the printer. The first is PRINTER, which again uses ON or OFF in the operand. Several of these directives can be used in a single program if you would like to list only a few subroutines. If you want to send the entire listing to the printer, it is probably easier to redirect the output from the command line. The other directive is EJECT. It takes no operand, and has no effect unless the output is going to the printer. In that case, it causes the system to skip to the top of a new page.

The last directive used to control the output is the TITLE directive. The TITLE directive doesn't need an operand, but it can take a string. If you use spaces in the string, it must be enclosed in quote marks. This directive causes the assembler to print page numbers at the top of each page. If a string was coded in the operand, the string is printed after the page number.

To review, the listing control directives, and their uses, are:

EJECT	new page on printer
ERROR	list/don't list errors
EXPAND	expand DC directives
LIST	list/don't list source
PRINTER	turn printer on/off
SYMBOL	list/don't list symbol tables
TITLE	place title at top of each page

Positioning Code

In the last section, we saw how to use the link editor to generate code that would run at various locations in memory. That's fine if the code needs to be assembled to run at several different locations, but can get to be a bore if the code must always be assembled to the same place. In that case, the ORG directive is more appropriate. The operand of the ORG directive is a constant, which can, of course, be coded as a hexadecimal or decimal value, or even as an expression. It should appear before the first START or DATA directive in the program. For example, if you are writing a program that uses the high resolution graphics page, you should code

```
        ORG     $4000
```

right after the KEEP directive.

The ORG directive can also be used before any subsequent subroutine to force it to start on a particular boundary. This is not a recommended practice. You can also use an ORG directive inside of a subroutine. If you want to use one of these unusual features, see page 147 for a more complete description of the ORG directive.

Looking much like an ORG directive, but with a very different purpose, is the OBJ directive. Like ORG, OBJ requires a constant as its operand. This directive is used when code must be located at one location, but will be moved to another before it is executed. For details, consult the reference manual.

Especially when time critical code is being written, it is sometimes nice to be able to force a segment to start on a particular memory boundary, generally a page boundary. (A page boundary occurs every 256 bytes.) To do that, include an ALIGN directive before the START directive for the segment. The operand for the ALIGN directive is a constant, and must be a power of two (1, 2, 4, 8, etc.). The link editor will insert zeros until the appropriate boundary is reached. The ALIGN directive can also be used inside of a segment, but in that case you must also have aligned the entire segment to a boundary equal to or larger than the boundary used inside the segment. The assembler will then insert the proper number of zeros to get the boundary.

The last directive used to position code is the MEM directive. Its operand is a pair of constants separated by a comma. The first constant must be less than or equal to the second one, and the directive must be found before the linker puts code in the address referenced by the first constant. Thus, the MEM directive, when used, should probably be in the very first segment. It has the effect of reserving the memory. When the linker starts to link a subroutine, it first checks to see if the subroutine will enter into a reserved memory area. If so, zeros are inserted until the reserved area has been passed. The only real use for this directive is when you are writing a graphics program that will start at $800. In that case, a

```
MEM     $2000,$4000
```

is appropriate to reserve the graphics page.

Controlling DC Code

The first directive that we will look at in this section actually affects more than just the DC directive. As you may know, the ASCII character set only uses the least significant seven bits of a byte. The most significant bit is usually off, but in the case of the character screen display used by the Apple, is on. Normally, ORCA produces characters with the high bit off for character DC directives and character constants in an expression. If you need characters with the high bit on, you can get them by coding

```
MSB     ON
```

Naturally, MSB OFF causes the high bit to be off.

If you are using the DC directive to generate floating point constants, you may notice that they are not in the same format as is used by Applesoft. If you need Applesoft format numbers, read up on the IEEE directive on page 143.

The MERR Directive

Errors found by ORCA have error levels. Appendix A, which explains the error messages and what to do about them, also has a discussion of what the different error levels mean.

If ORCA finds an error, it will normally not go on to link edit and execute a file; however, you may have a situation where the error is expected, and is in fact all right. If that is the case, the MERR directive can be used to tell the system that it is all right to go on. The operand is a constant that tells the system the highest error level that it should ignore. Normally, of course, this is zero.

Using the 65C02

If you have an Apple //c, an upgraded Apple //e, or if you have changed the CPU in an older Apple, you have the 65C02 instruction set. The ORCA assembler is quite capable of generating code for that chip, whether or not you actually have one. The 65C02 directive has an operand of ON or OFF, defaulting to OFF. If you code

```
65C02 ON
```

you will enable the new instructions and addressing modes found in the 65C02. Otherwise, the assembler flags these as errors so that you know that your code is safe for a 6502 machine, even if yours happens to be a 65C02 machine.

Note that there are, in fact, two versions of the 65C02. The standard instruction set was augmented by Rockwell when they produced their version, to include extra thirty-two bit manipulation and bit test instructions. These instructions are not in the standard instruction set of the 65C02, and do not appear on the 65816, so we recommend that you never use them, even if you have the Rockwell version of the 65C02. They are not included in the assembler. If you need them, they are provided as macros in the M6502.65C02 macro library. The next chapter tells you how to use the macro libraries.

Using the 65816 or 65802

A new sixteen bit version of the 6502 is now available. In one package (the 65802) the chip can still only address 64K of memory, but it can be plugged right into your older Apple. The instruction set is a superset of the 65C02 (which is in turn a superset of the 6502), so all of your old software will still work. The other version uses the same instruction set, but can address 16 megabytes. This is the version used in the Apple IIGS. ORCA can assemble code for the new CPU. To enable the new op codes and addressing modes, type

```
65816 ON
```

Naturally, 65816 OFF disables the new features. When off (the default mode) the assembler identifies all extensions as errors so that you can still generate code for the 6502 and be sure that you have not used the new features.

In addition, it is necessary to be able to tell the assembler whether immediate addresses will be eight or sixteen bits. The LONGA and LONGI directives are used to do this job. If you plan to develop code for the 65816, read about these directives starting on page 144.

Note that this version of ORCA contains a linker that generates binary executable files. This file type can be executed from an Apple IIGS, but only from ProDOS 8. If you are writing programs to run under ProDOS 16,

you need the Apple IIGS version of ORCA. Programs written using this version of ORCA can be assembled with the version for the Apple IIGS.

Chapter 7: Using Macros

Tools of the Trade

The ORCA assembler comes with a large set of predefined macros. This collection of macros is so large and complete that most people will probably never need to write a macro of their own. In this chapter we will look at how you can use the macros in ORCA (or any other macro library) even if you never intend to write one of your own.

Lets start by getting used to the the tools that we will use. We will do this by writing a short program to read two four byte integers, do the basic four math operations on them, and quit. To do this, we will use the macros in the following list. Although it is not strictly necessary, you may want to stop and read about these macros in the reference manual before we go on. The page number that each is described on is in the third column, the name of the macro is in the first, and a short description of what it is used for is in the second column.

GET4	read a four byte integer	199
PUT4	write a four byte integer	211
PUTS	write a string	211
PUTCR	write a carriage return	213
ADD4	add two four byte integers	183
SUB4	subtract two four byte integers	191
MUL4	multiply two four byte integers	187
DIV4	divide two four byte integers	185

Let's just dive right in, and write the program. After it is up and running, we will do some things to explore what is really going on. Enter the following program, exactly as it appears, and save it as C4. Don't try to assemble it right away!

```
            KEEP  CALC4
            MCOPY C4.MACROS
*****************************************
*
*  Four Byte Integer Calculator
*
*****************************************
*
C4          START
;
;  Read the two input numbers.
;
            PUTCR
            PUTS  #'First number:  '
            GET4  NUM1,CR=T
            PUTS  #'Second number: '
            GET4  NUM2,CR=T
            PUTCR
;
;  Do the calculations and print the results.
;
            ADD4  NUM1,NUM2,NUM3    addition
            PUTS  #'Sum:  '
            PUT4  NUM3,#1,CR=T

            SUB4  NUM1,NUM2,NUM3    subtraction
            PUTS  #'Difference:  '
            PUT4  NUM3,#1,CR=T

            MUL4  NUM1,NUM2,NUM3    multiplication
            PUTS  #'Product:  '
            PUT4  NUM3,#1,CR=T

            DIV4  NUM1,NUM2,NUM3    division
            PUTS  #'Ratio:  '
            PUT4  NUM3,#1,CR=T
            PUTCR
            RTS

NUM1        DS    4
NUM2        DS    4
NUM3        DS    4
            END
```

The first thing that you will probably notice is that there aren't many familiar instructions. In fact, only one assembly language instruction appears in the entire program. All of the other lines of code are macro calls. Like all other statements, macros have a label field, op code field, and operand. We didn't happen to use the label field, but it works just like the label field of any other statement. Macro operands vary a great deal. Later, we will look at the macro reference manual and see how to tell from it what the operand format is.

If you ignored the earlier warning and tried to assemble this program, you got a file not found error. That's because of the MCOPY directive. The MCOPY directive tells the assembler where to go to look for macros. Since we haven't created the file yet, you get an error.

It is possible to tell the assembler to use the macro libraries that come with ORCA directly, but that is very inefficient. It takes a long time to switch between macro library files, and the macros we need are in two different library files, M6502.I.0 and M6502.INT4MATH. Instead, we will build a macro library file just for this program which has only those macros that we actually need. To do that, we will use a utility called MACGEN. To run it, type

```
MACGEN C4
```

where C4 is the name of the program you just typed in. (If you called it something else, use that name instead.) MACGEN will start out by scanning your program and building a table of all of the macros that you need. Each time it finds a new macro, it prints a dot. Once it has scanned your entire program, it will print a list of the macros that you need and ask for an output file name. You should respond with C4.MACROS, the same name that was used in the MCOPY directive. Next, you will be asked for the name of a macro library to search. The easiest thing to do (but not the fastest) is to use the fact that the program supports wildcards and give it the name of the macro library disk. If you are using the system the way it was shipped to you, you would type /MACROS/=. After a bit of whirring, you will have a macro file.

Now you can run the program. You will notice that it takes a while to assemble the program, even though it's fairly short. The reason is easiest to see at the end of the assembly: the lines generated message tells you how many lines the assembler created while expanding the macros. Also, note all of the extra subroutines that the link editor finds. These all come from

the subroutine libraries. If you have ever looked at what it takes to do these math operations and input and output in software, you can appreciate the work that you don't need to do! Take a moment to play with the program. You can enter some fairly large integers! If they aren't large enough for you, you might want to convert the program to use eight byte integers by changing all of the 4 characters in the macro names to 8 and enlarging the DS areas to 8 bytes each.

Well, that's your first program that uses macros. Some minor points should be mentioned. First, a macro file can be as large as your disk. If for some reason you need to use more than one macro library at a time, you certainly can - up to four can be used, and the MDROP directive lets you get rid of those you don't need anymore. Multiple macro libraries for a single program are not, however, recommended. To find out more, read about MDROP, MLOAD and MCOPY starting on page 145.

You can look at the code generated by the macros by using the GEN directive at the top of your program. If you do that, you might also want to use the -C flag on MACGEN. Normally, there is no reason to expand the macros, so we won't cover it in detail here. See page 80 for a full description of MACGEN, and page 143 for a description of the GEN directive.

The Macro Library

There are really three reference sections in this manual. The first two cover the operating system, utilities and assembler, and you have probably already used them. The last is the macro reference manual. It starts on page 175, and covers the operand and data formats, as well as the macros used to manipulate the data. It is divided into five major sections: Input and Output, ProDOS, Mathematics, Graphics and Miscellaneous.

As you learn the macros, you should think of them as a new, larger instruction set. Keep in mind that it took you some time to learn the instruction set of the 6502, and it will also take time to learn about the macros in the macro library. Start by reading the introductory material about addressing modes and data formats, then scan the reference manual to look for macros that fit your needs.

If you need a macro that you suspect exists, or if you have forgotten the name of a macro, you will find a list of them on the reference card. After you have the name, you can get a page number from the index and read up on the macro in the reference manual.

Keyword Parameters

In the example program that we wrote earlier, we used two kinds of parameters in the operands of the macros. The most common kind is the positional parameter, which works like the operand of assembly language instructions and assembler directives. If you look in the reference manual, each macro descriptions starts with an example of the macro. The number of parameters that are allowed can be found by counting the number of parameters in the example. Note that not all allowed parameters are actually required. The description of the ADD4 macro, for example, points out that the third parameter is optional. If you code

```
ADD4  NUM1,NUM2
```

the two numbers are added and the result saved at NUM1.

Occasionally, it is easier to remember the name of a parameter than it is to remember its position. That is when keyword parameters come in. The example macro serves a double purpose: in addition to telling you what position the parameters are in, the names used also indicate the keyword that is associated with each parameter. Keyword parameters are specified as the parameter's keyword followed by an equal character and the string to set the parameter to. In our sample program, we used a keyword parameter on the PUT4 macro. If you check the reference manual, you will find out that if you assign anything to the CR parameter, the macro will output a carriage return after writing the integer. Rather than remember what the position of the CR parameter is, we coded a CR=T in the operand.

So how do you tell the difference between a keyword parameter and a positional parameter? Actually, you don't. Any parameter can be set using either position or a keyword, so you can use whichever method you wish. In fact, as you can see from the PUT4 macro, you can even use positional and keyword parameters together. Keep in mind, though, that keyword parameters take up a space. Coding

```
PUT4  NUM1,CR=T,#4          Wrong
```

would not right justify the number in a four byte field, since the format value must be the second parameter. The proper way to code the macro would be

```
PUT4  NUM1,#4,CR=T          Right
```

Chapter 8: Writing Macros

As was pointed out in the last chapter, you do not need to be able to write a macro in order to use them, and in fact, since so many are provided, many people will never need to know how to write a macro of their own. For that reason, you are urged not to try until you are fairly familiar with the system as a whole.

But of course you're curious, and want to give it a try anyway. Well, have fun! This chapter is a tutorial on writing macros. The first step in successfully writing macros is to get a perspective on the task. To write macros, you will need to learn a new programming language. This is a fairly unusual language, since it is used to program the assembler itself.

Like other programming languages, the one you are about to learn has variables, which are called symbolic parameters. There are three data types: arithmetic variables, boolean variables, and string variables. You can define arrays of variables, pass parameters, do operations on the variables, and assign the results of those operations to the variables. The input to this language takes the form of source statements in the assembly language program. The ultimate output is also in the form of source statements to the assembler.

So, that's what this chapter is all about. We will start by learning how to define a subroutine in our new language.

MCOPY, MACRO and MEND

Of course, you remember right away that in the last chapter we said that macros are not subroutines, right? Well they aren't, at least not in the sense of a subroutine in your program. But a macro does in fact serve the same purpose in the conditional assembly language as subroutines in a program do. We define a named sequence of instructions which accepts parameters, and use this named sequence of instructions in our program. The output from the macro is a set of assembly language source statements, which the assembler then assembles, adding code to our program.

Each macro definition has a MACRO directive as the first statement. The macro directive serves the same purpose as the START directive does for a code segment: it marks the beginning of a new macro, just as START marks the beginning of a new segment. The last line of every macro is an MEND

directive, which marks the end of the macro. Neither of these directives needs an operand, and neither can make any use of a label.

The line right after the MACRO directive describes the parameters of the macro to the assembler. It is called the macro model line. The op code on that line is the name of the macro, and is the same name that is coded in the program to invoke a macro. This line is required, and must be the line immediately after the macro directive. Leaving a blank line between them, for example, would cause the macro to not work correctly.

Finally, the lines between the macro model line and the MEND directive are called macro model statements. These lines are the ones that are sent to the assembler when you use the macro in your program.

Lets stop for a minute - there were a lot of new terms in the last few paragraphs. To firm up those ideas, let's write a very simple macro. The Apple Reference Manual lists a series of entry points into the F8 ROM which can be used to perform special functions. One of these, called PRBYTE, prints the contents of the accumulator on the screen. It is located at $FDDA. Of course, you remembered that, right? Well, I didn't. But I did remember that it was called PRBYTE, so I can define a macro to do the JSR and forget about the number. The resulting macro produces a single source line, a JSR to $FDDA. It is shown below.

```
MACRO
PRBYTE
JSR   $FDDA
MEND
```

We enter this macro into the editor, just like we would enter a program. Afterwards, you can save it in a file called TEST.MACROS, and then type the following program into a new file.

```
          KEEP  TESTIT
          MCOPY TEST.MACROS
TEST      START

          LDX   #0
LB1       LDA   0,X
          PRBYTE
          INX
          BNE   LB1
          RTS
          END
```

When you run this program, it will print the contents of zero page onto the screen. Note that the hexadecimal value $FDDA doesn't show up anywhere in the program we wrote - the assembler pulls in the JSR instruction from a separate macro file.

All right, we admit that you could do this with an equate. You could even write a lot of equates and put them into a file that can be copied into your program. But our macros will get much more capable!

There are two directives that we will not look at here which are also a part of the macro language. MEXIT allows early exit from a macro. It is discussed on page 171. MNOTE is used to generate error messages in a macro, and can be found on page 171.

Basic Parameter Passing

In the last section, we found out how to write a macro that would do a simple substitution of code. Now, we will look at a way to cause one macro to generate different code when it is used in two different places. To do this, we need to learn how to pass parameters to macros.

The first line of the macro, like most lines, has a label field and an operand field. In the last section, we didn't put anything there. In fact, we can't put the normal kinds of things in those fields. Instead, the label field of the statement must contain a symbolic parameter, if it has any thing at all, and the operand field field can only contain symbolic parameters, separated by commas. Symbolic parameters are the variables of the conditional assembly language. They start with an &, and are followed by a label. Symbolic parameters defined by passing a parameter to a macro are always string variables. The string they contain is the string that was coded in the macro call. Once defined, a symbolic parameter can be used anywhere in

an assembly language statement. The assembler always starts by replacing any symbolic parameters with the equivalent string of characters.

As an example, we will write a simple macro to add two two byte integers. We will pass three parameters, each of which is the name of a label where a two byte integer can be stored. The macro will add the contents of the first two locations and place the result in the third. We will also define a label, and place it on the CLC instruction. The macro looks like this:

```
          MACRO
&LAB      ADD   &NUM1,&NUM2,&NUM3
&LAB      CLC
          LDA   &NUM1
          ADC   &NUM2
          STA   &NUM3
          LDA   &NUM1+1
          ADC   &NUM2+1
          STA   &NUM3+1
          MEND
```

This macro is a little more complicated than our first, and you may want to see just what it really produces. That is, in fact, a good idea. After writing any new macro, you should expand it several different ways and look at the source lines it produces. Normally, the assembler doesn't put the lines generated by a macro in the listing, but you can force it to by placing a GEN ON directive in the program. If the above macro is in a file called TEST.MACROS, then the following program will show you what happens.

```
          MCOPY TEST.MACROS
          GEN   ON
TEST      START

          ADD   I,J,K
LB1       ADD   K,J,I

I         DS    2
J         DS    2
K         DS    2
          END
```

There is no need to link or execute the program, since all we want to do is look at the lines produced by the macro. If you don't try to link the program, you really don't even need the definitions of I, J and K. Pay

special attention to what happens to the label, LB1. In fact, take a moment to change the macro, placing the &LAB parameter on the last ADC, instead of on the CLC, and see what happens. Using a label on a macro does not define the label. It only gets defined if the macro places the label on a statement that it produces.

Defining Symbolic Parameters

The next four sections introduce the instruction set of the conditional assembly language. Although we will look at some examples of individual statements, we will have to put off until later a really meaningful example, since some parts of all of the next few sections are needed.

When we defined the ADD macro in the last section, we also managed to define four symbolic parameters at the same time. In this section, we will look at a more direct way of defining a symbolic parameter. Six directives come to our aid. Let's start by defining a string (or character) type symbolic parameter. The op code is LCLC, or local character, and the operand is the name of the symbolic parameter we want to define. There is usually no label.

```
        LCLC  &STRING
```

After the assembler encounters this line, the symbolic parameter &STRING is defined. It also has an initial value, the null string (a string with no characters). Strings have a length as well as a value, and at this point the length of &STRING is zero. A string variable can hold up to 255 characters.

To define an arithmetic symbolic parameter called &NUM and a boolean symbolic parameter called &LOGIC, we would use

```
        LCLA  &NUM
        LCLB  &LOGIC
```

Both are initialized to zero. Arithmetic symbolic parameters can contain any four byte signed integer value, while boolean symbolic parameters can take on any value from 0 to 255. When used in a logical expression, zero is treated as false, and any other value as true.

The symbolic parameters defined so far are all local symbolic parameters. If defined in the program itself, they go away when the END directive is encountered. If used in a macro, they vanish when the MEND directive is

found. Note, however, that if a macro contains a macro call, the symbolic parameter stays around for the second macro. For example, the following code will work (although it is meaningless):

```
MACRO
MAIN
LCLA  &NUM
NEWMAC
STA   &NUM
MEND

MACRO
NEWMAC
LDA   &NUM
MEND
```

Macros defined by the first line of the macro are also local.

There is another way to define a symbolic parameter so that it is global. The following lines redefine the three symbolic parameters as global ones.

```
GBLA  &NUM
GBLB  &LOGIC
GBLC  &STRING
```

Global symbolic parameters still vanish at the end of a segment, when the END directive is encountered, but they do not go away when a macro expansion has finished. This fact lets you create macros that pass information to one another by defining global symbolic parameters and assigning values to them.

Symbolic parameters, as we said, can be subscripted. To define an array of symbolic parameters, place the maximum size of the array after the name of the symbolic parameter, in parentheses. For example,

```
LCLC  &STRINGS(10)
```

defines an array of ten strings. Using &STRING(4) in a statement would cause the fourth string in the array to be used. The number of array elements that can be declared varies with the type of the symbolic parameter. See the reference section for details.

Changing and Using Symbolic Parameters

The values of symbolic parameters are changed using set symbols. The set symbol is a directive that is logically equivalent to the assignment operator in most languages, but unlike most languages, the operator itself is typed. This means that the directive used to assign a value to an arithmetic symbolic parameter is different from the one used to assign a string to a character symbolic parameter. The set symbol directives are SETA, SETB and SETC. All three directives must have a symbolic parameter in the label field, and it is that symbolic parameter that is changed.

The SETA and SETB directives both use a constant expression in the operand field. The operand is evaluated to give a fixed integer result. In the case of the SETA directive, the result is assigned directly to the arithmetic symbolic parameter, while in the case of the SETB directive, the result is first taken mod 256, yielding a result between 0 and 255. Generally, logical expressions are used in the operand of a SETB directive, such as CONST<=0. Logical expressions always produce a result of zero or one, which correspond to false and true, respectively.

The operand of a SETC directive is a string. It must be enclosed in quote marks if the string contains spaces, starts with a quote mark, or is a string operation. Two strings can be concatenated by the SETC directive: to do that, simply separate them with a + character.

Some examples of set symbols are shown below.

```
&NUM      SETA  CONST-CONST/65536*65536
&NUM      SETA  &NUM-1
&LOGIC    SETB  &NUM>0
&LOGIC    SETB  1
&STRING   SETC  &STRING+&NUM
&STRING   SETC  10.5
&STRING   SETC  'Here''s a quoted string'
&ARR(4)   SETA  16
&ARR(&NUM) SETA &ARR(4)
```

The examples above also point out how array elements are set and used, as well as the fact that a symbolic parameter can be used to specify an array subscript. There is one more fine point about symbolic parameters that we should point out, and that is the use of the dot operator. If a symbolic parameter is followed immediately by a dot, the dot is removed from the line during expansion of the symbolic parameter. To see why this is useful,

let's look at a somewhat contrived example. Let's assume that in a macro you will be doing an operation on the X register, but that you do not know if it will be a load or store. The first two characters of the operand are contained in the symbolic parameter &OP. Then the dot operator is used to indicate that the X is not a part of the symbolic parameter name. If &OP contains the characters LD, then the following line creates a LDX operation.

```
        &OP.X ADDR
```

Unfortunately, the dot operator is encountered far more often in logical expressions. The expression &LOGIC.AND.&LOGIC2 gives a syntax error: it must be coded as &LOGIC..AND.&LOGIC2.

If you have tried any of the above directives in a test program, you may have noticed that they weren't printed. That is because once a macro is developed, it is rare to want to see all of the conditional assembly line that go into generating the code. If you need to look at those lines, you should make use of the TRACE directive, described on page 150.

String Manipulation

Two directives greatly increase your ability to work on strings. These are the ASEARCH directive, which lets you search one string for occurrences of another, an AMID, which allows selection of a small number of characters from a string.

The ASEARCH directive has three operand fields separated by commas. The first is the string that you want to search, the second is the string to search for, and the last is the position in the string to begin the search. The result is a number, so the label field must contain an arithmetic symbolic parameter. It is set to the position in the string where the search string was found, or zero if it wasn't found. The comment fields in the following examples tell what values would be assigned.

```
&NUM    ASEARCH 'TARGET STRING',RG,1      3
&NUM    ASEARCH 'TARGET STRING',R,1       3
&NUM    ASEARCH 'TARGET STRING',R,4       10
&NUM    ASEARCH 'TARGET STRING',Z,1       0
&NUM    ASEARCH 'TARGET STRING',R,11      0
```

The AMID directive also takes three operands, separated by commas. The first is the string, the second is the first character to select, and the third is the number of characters to select. It is legal to ask for characters that are

outside of the range of the string, in which case a character is not returned. The label field of the directive must contain a character type symbolic parameter. The examples below show the resulting string in the comment field.

```
&STRING AMID   'TARGET STRING',2,3       ARG
&STRING AMID   'TARGET STRING',12,3      NG
&STRING AMID   'TARGET STRING',20,3      null string
```

There is one more directive that can set the value of a symbolic parameter. It is called AINPUT, and it lets the assembler get a response from the keyboard. It is discussed on page 165.

Conditional Assembly Branches

Perhaps the most important capability in the conditional assembly language is the ability to branch, thus skipping code, or looping over it several times. To use a branch, we must first have something to branch to. Conditional assembly labels take the form of a period in column one, followed by a label. These statements are called sequence symbols, and are treated as comments unless the assembler is looking for a place to branch to.

There are two branching directives. The first is called AGO. Its operand is a sequence symbol. It is an unconditional branch. You might try assembling a short test program with the following code to convince yourself that it works.

```
         AGO   .THERE
;This comment will not be in the final program
         LDA   #4       ...and neither will this LDA
.THERE
```

The other branch is a conditional branch, similar in function to the BASIC construct, IF condition THEN GOTO label. The operand is a logical expression followed by a comma and a sequence symbol. If the expression evaluates to 0, it is false, and if it evaluates to 1 it is true. Both of the following code fragments produce the same result, four ASL A instructions. Again, you should try them out in a short sample program.

```
          LCLA  &N
&N        SETA  4
.TOP
          ASL   A
&N        SETA  &N-1
          AIF   &N>0,.TOP

&N        SETA  4
.TOP2
          ASL   A
&N        SETA  &N-1
          AIF   &N,.TOP2
```

There are some tricks and another directive that we won't go into here. The ACTR directive lets you bypass the built in protection against infinite loops, and is described on page 163. Efficiency considerations for writing faster branching code are discussed on page 164.

Attributes

Occasionally, it is nice to know more about a label or symbolic parameter than what it's value is. That is where attributes come in. Attributes give you a way of asking questions about the label or symbolic parameter. They take the form of a letter, a colon, and the name of a label or symbolic parameter. Attributes are used like functions, being mixed into an expression as if they were an integer constant.

The first attribute that we will look at is the count attribute. It is used to tell if a label or symbolic parameter has been defined, and if so, how many subscripts are available. The count attribute of an undefined label or symbolic parameter is zero. The count attribute of a defined label, or a defined symbolic parameter that is not subscripted, is one. The count attribute of a subscripted symbolic parameter is the number of subscripts available. The count attribute is used in the following loop to initialize a numeric array for a symbolic parameter that may or may not be defined.

```
          LCLA  &N
&N        SETA  C:&ARRAY
          AIF   &N=0,.PAST
.TOP
&ARRAY(&N) SETA &N
&N        SETA  &N-1
          AIF   &N,.TOP
.PAST
```

It may seem like poor programming is the only case where you would not know if a symbolic parameter had not been defined, but there are in fact two very common uses for the count attribute. The first is when a macro will define a global symbolic parameter to communicate with any future versions of itself. In that case, the macro can test to make sure the parameter has not been defined already. The following macro uses this fact to define a sequence of integers. You don't need to count the macros, just put in a handful - they count themselves.

```
          MACRO
&LAB      COUNT
          AIF   C:&N>0,.PAST
          GBLA  &N
.PAST
&N        SETA  &N+1
&LAB      DC    I'&N'
          MEND
```

The second use is to check and make sure a parameter was passed. We could modify our original ADD macro so that if the last parameter were omitted, the result could be stored in the first location. This makes use of the fact that the assembler doesn't define an operand unless the macro call statement uses it. The new macro would look like this:

```
          MACRO
&LAB      ADD   &NUM1,&NUM2,&NUM3
          AIF   C:&NUM3,.PAST
          LCLC  &NUM3
&NUM3     SETC  &NUM1
.PAST
&LAB      CLC
          LDA   &NUM1
          ADC   &NUM2
          STA   &NUM3
          LDA   &NUM1+1
          ADC   &NUM2+1
          STA   &NUM3+1
          MEND
```

It is even possible to subscript a symbolic parameter from a macro call. For a full discussion of that topic, see page 158.

The next symbolic parameter is the L, or length, attribute. The length attribute of a label is the number of bytes created by the line where the label was defined. This makes counting characters very easy! The DW macro from the macro library takes a string and precedes it by a one byte integer containing the number of characters in the string. It is a very simple macro:

```
          MACRO
&LAB      DW    &STR
&LAB      DC    I1'L:SYSA&SYSCNT'
SYSA&SYSCNT DC C'&STR'
          MEND
```

It also demonstrates the use of the &SYSCNT symbolic parameter. This symbolic parameter is predefined by the system, and is incremented once at the beginning of each macro expansion. It is what prevents two occurrences of the DW macro in the same code segment from creating a duplicate label. You should try the above macro in a short sample program to see how this works.

The length attribute of an arithmetic symbolic parameter is four. The length of a boolean symbolic parameter is one. The length of a string symbolic parameter is the number of characters in the string.

There are two more attributes that we will not discuss here, since the are rarely used. The type, or T attribute, is used to tell what kind of statement

generated a label. The S, or settings attribute, is used to detect the current setting of the assembler flags. See page 161 for a discussion of these attributes.

This concludes the tutorial of the conditional assembly language. Like all languages, it will seem strange until you have used it for a while. Practice is the only way to overcome that difficulty. If you would like to see some example of how the conditional assembly language is used, look at any of the macros in the macro library.

Chapter 9: Writing Shell Programs

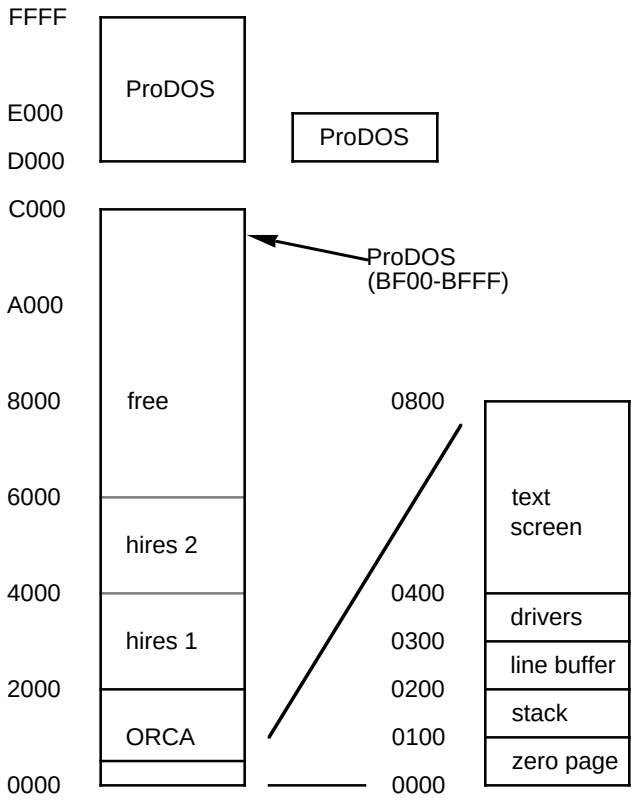
In this chapter, we will take a look at how to write a program that runs under the ORCA shell. What environment your program will run under is really up to you. There are three common choices: system programs, which execute directly from ProDOS when the computer is turned on; programs that are executed from BASIC.SYSTEM; and programs that are executed from the ORCA shell.

If you are writing a system program, then none of this affects you. This chapter points out the differences between programs designed to run under BASIC.SYSTEM and those that run under the shell. Many programs can actually be written which will run under either environment. When writing your program, the things that you will need to consider are: what memory is available, how input will be received, and if it runs under the ORCA shell, whether the program should be installed as a utility. The sections in this chapter deal with those issues.

Memory Map

The memory map for the shell is actually quite simple. Since the editor, assembler and linker are loaded from disk when needed, they are not in memory when your program is running. The only memory used is from $800 to $1FFF, where the ORCA/HOST overlay processor and device drivers are located. If you are using a non-standard clock card driver, ORCA.SYSTEM installed it in page 3 during the boot process, so you should not disturb that area. Finally, zero page locations $FA to $FF are used by the system for entry pointers, and must not be disturbed.

If you make a call to the shell from your program, certain other zero page locations may be changed. These are $80 to $F9. While these may be used during the call, ORCA/HOST does not depend on the values between calls. Thus, you can use this area for temporary variables safely.



The result of this memory usage is that programs designed to run under ORCA, like ProDOS system programs, will normally start at $2000. Certain graphics applications might start at $4000 or $6000 to avoid the graphics pages. Your program can use the memory from $2000 through $BEFF. The area from $BF00 to $BFFF is used by ProDOS.

When using zero page, be sure and take into account the memory used by the F8 ROM, ProDOS, and if you are using calls into it, the BASIC interpreter. See the Apple // reference manual and the ProDOS reference manual for exact locations and uses.

Reserving Memory

Knowing what memory is available is only part of the story. The other part is claiming it so that ProDOS and ORCA/HOST do not use it. This is

where the ORCA shell and BASIC.SYSTEM differ the most. Under BASIC, when your program executes, all of the memory is marked as available. Your program must then reserve the memory that it occupies to keep itself safe. When your program executes from ORCA, just the opposite happens. ORCA reserves the memory where your program is located. Since ProDOS expects that memory that it will write to is free, you must deallocate memory within your program if you want to read to it. The RELEASE macro can be used to do this. When you want to reserve it, or to find a piece of memory of a particular size, use the RESERVE and FINDBUFF macros.

If you are unfamiliar with the way memory is handled under ProDOS, refer to the ProDOS technical reference manual. For more information on the macros mentioned above, see the reference sections on the macros.

The Console Driver

BASIC.SYSTEM used a simplified console driver that is limited to writing ASCII characters (with the high bit set). Like Apple Pascal, ORCA uses a more sophisticated console driver. It is capable of clearing the screen and homing the cursor when a form feed character is written, of positioning the cursor, and so on. The functions available are accessed using macros. The important point to realize is that if you do anything but write text to the screen, your program cannot work from both ORCA and BASIC.SYSTEM.

Passing the Command Line

One of the advantages to writing a program that executes under ORCA is that the program can read the command line to find out what the inputs are. Doing so is a key element in making a program run smoothly in the ORCA environment. It also happens to be easy.

Before ORCA executes your program, it takes the command line and stores it in an internal buffer. The command name and leading blanks are removed, along with I/O redirection commands. (I/O redirection is handled automatically. Your program does not need to do anything special to make it happen.) Right before calling your program, the address of the command line is placed in zero page locations $2 and $3. The string is a sequence of ASCII characters ending with a return character.

The program below illustrates these concepts. It is called ECHO. What it does is to write the characters following the command name to standard out.

```
          KEEP  ECHO
***********************************************
*
*  ECHO - Echo the command line
*
***********************************************
*
ECHO      START
RETURN    EQU   $0D           RETURN key code
LINE      EQU   2             addr of the line pointer

          LDY   #0
LB1       LDA   (LINE),Y
          JSR   COUT
          INY
          CMP   #RETURN
          BNE   LB1
          RTS

COUT      JMP   ($36)
          END
```

Installing a New Utility

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

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

1. Place the program in the utility prefix. As shipped, this is the /UTILITY disk, but you may have moved it to your hard disk if you are using one.

2. Add the program name to the command table. The command table is in the system directory, where the editor is located. It is called SYSCMND. The command table is a text file, and can be changed with the editor. Simply edit it, and add the name of your program to the list of commands you see. After at least one space, type a U, which indicates that the command is a utility.

Be sure and put the command in the correct location. For our example utility above, we might not want to put it before EDIT, since we would no longer be able to use E followed by a right arrow to abbreviate EDIT. Instead, place it after EDIT. To add ECHO as a utility, then, add the line

```
ECHO      U
```

right after the EDIT line in the command table.

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

3. If you would like to have online help for the command, add a text file to the HELP directory in the utility prefix. The name of the file must be the same as the name of the utility.

Installing a Compiler or Editor

Since the ORCA development environment is already broken up into small pieces, it is easy to replace one of those pieces, or to add one to those that exist. Changing the ORCA editor is the easiest thing to do. Simply place the new editor in the system prefix and call it EDITOR. When it is called by the system, the editor should do a GET_LINFO call using the macro supplied with ORCA. Two fields are of interest here. The first is the SOURCE file, which is the full or partial path name of the file to edit. The second is ORG, which is the displacement into the file where the cursor should be placed. Usually, this is zero, but if a language aborted with an error, it can call the editor with the location of the offending line. To return to the shell, the editor must use the SET_LINFO call.

Installing a compiler is more difficult, but certainly easy compared to writing one! Again, the GET_LINFO call is used to fetch input parameters, and the SET_LINFO call is used to return to the shell. The section that describes these two shell calls explains what the compiler will see.

If you are installing a compiler in ORCA, please get in touch with us so that we can assign a unique number to your language. This prevents conflicts with other languages that may be placed on the system in the future.

Chapter 10: Introduction

Using The Manual

This section is the technical reference manual for the ProDOS ORCA operating environment, version 4.1. It contains all of the technical information needed to use the ORCA/HOST operating system, command processor, and system utilities. This information is presented in a detailed, technical manner, laid out for easy reference.

This is not the manual to read during a relaxing evening in front of the fireplace; it is intended for detailed reference only. The first section, the User's Manual, is what one should read to get familiar with the system. As the user's manual is read, it will refer to this manual for detailed material. After reading the user's manual, it is this manual which will be the easiest place to look up details, find command references, or explore the meaning of error messages.

Introduction

ORCA/HOST is a collection of programs which provide an operating environment for the ORCA family of languages. Its main modules are described below. Later chapters give details for each module.

ORCA.SYSTEM

The ORCA system program is a short initialization and boot up program. It conforms to ProDOS conventions for system programs. It installs non-standard clock card drivers in ProDOS (if any), sets the date, and loads and executes ORCA/HOST.

ORCA.HOST

The ORCA/HOST program is an operating system supplement to ProDOS. One of its main functions is to provide a link between ORCA and user programs which allows them to ignore all hardware dependent aspects of the Apple. ProDOS provides a good user interface for disk drives and clock cards, so ORCA/HOST does not deal with these areas; instead it concentrates on the keyboard, the CRT display, and printers.

A second function of ORCA/HOST is to provide an advanced inter-process communication facility which allows ORCA language programs to communicate with each other. This is what allows the ORCA RUN command to switch between various assemblers and compilers, automatically invoke the link editor, and execute a program, all with a single command. It also provides input and output redirection and command (or EXEC) files.

MONITOR

The monitor houses an advanced command processor. There are three distinct types of commands supported by ORCA; the resident command, the utility, and the language. Resident commands are built right into the monitor, and execute immediately after the command is typed. Utilities are syntactically equivalent to resident commands, but are in fact separate programs. When a utility command is entered, the monitor looks for the utility on a special directory and executes the program. When it finishes, the utility returns control to the monitor. The last type of command is the language, which every ORCA source file has. It is the language that tells the system what compiler or assembler to use when translating from source to object code.

Utilities

The structure of the command processor allows for a large number of external utilities - far more than could be supported if all commands were resident in the MONITOR file. Another less obvious advantage is that utility commands can be added to those currently available. The utilities provided with the system are described in Chapter 11. Three complex utilities of special importance are described in Chapters 12 through 14; these are the system text editor, link editor and disassembler.

System Configuration

System configuration is basically a very automatic process. The introduction to the user's manual describes this process from the User's viewpoint; this section gives a more technical description of exactly what happens.

On the distribution copy of ORCA, the first system program that ProDOS will find is called INIT.SYSTEM. This is a short program which prompts

the user to answer two questions: what clock-calendar card (if any) is being used, and what 80 column board is being used. These are the only two hardware options currently supported which require changes in code. All clock software is handled by ORCA.SYSTEM by installing a clock driver into ProDOS. Eighty-Column board drivers are contained in ORCA.HOST. As a result, making the required changes to the code is a simple matter of deleting all of the unwanted versions of ORCA.SYSTEM and ORCA.HOST, and renaming the proper versions. Upon completing its task, INIT.SYSTEM deletes itself and transfers control to ORCA.SYSTEM.

Once the system is configured for the particular hardware in use, it may be desirable to change the default locations where the system will look for various files, add or rename commands in the command table, change the printer initialization string, etc. These actions are handled by modifying the command table, called SYSCMND, and by using the SET command. The format for the command table is described with the COMMANDS command. The options that can be changed with the SET command are described with that command.

Chapter 11: The Command Processor

Types of Commands

Commands in ORCA 4.1 can be subdivided into three major groups. All are entered from the keyboard the same way.

Built in Commands

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

Utilities

ORCA 4.1 supports commands in the command table which require a program external to the command processor to execute. An example of this type of command is PEEK, which is a separate program under ORCA 4.1. The programs to perform these commands are contained on a special directory known as the utility directory. The command processor must first load the program that will perform the required function, so the utility directory must be on line when the command is entered. The command will also take longer to execute, since the operating system must load the external program, execute it, and then reload the command processor.

The utilities themselves must all reside in the same subdirectory so that the command processor can locate them. The name of the utility is the same as the name of the command used to execute it. Utilities are responsible for parsing all of the input line which appears after the command itself, except for input or output redirection. The address of the command line (with the command and I/O redirection parameters stripped) is passed to the utility in locations $0002 and $0003. An RTS at the end of the utility returns control to the command processor.

Language Names

The last type of command is the language name. All source files are stamped with a language, which can be seen when the file is cataloged under ORCA. (ProDOS will not show the language names.) There is

always a single system language active at any time when using ORCA. New files will normally be stamped as ASM6502.

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

The following table shows a partial list of the languages and language numbers that are currently assigned. CATALOG and HELP will automatically recognize a language if it is properly included in the command table. ProDOS has a special status: it is not truly a language, but indicates to the editor that the file should be saved as a standard ProDOS TXT file. Language numbers are used internally by the system, and are generally only important when adding languages to ORCA. They are assigned by the Byte Works, Inc. Those marked with an asterisk are in the command table shipped with the ORCA assembler.

	language	number
*	PRODOS	0
*	TEXT	1
*	ASM6502	2
	ASM65816	3
	BASIC	4
	PASCAL	5
*	EXEC	6

Program Names

Anything which cannot be found in the command table is treated as a path name, and the system tries to find a file that matches the path name. If a binary file is found, that file is loaded and executed. If a source file with a language name of EXEC is encountered, it is treated as a file of commands, and each command is executed, in turn. If the file is a system file, it is executed, but cannot return to ORCA. See Command Files, later in this chapter, for details.

Entering Commands

Under ORCA 4.1, commands must be spelled out in full, as with ProDOS. To avoid writer's cramp (typer's cramp?) one may hit the right arrow key at

any time while entering a command. If the letters typed so far are a part of a command found in the command table, the rest of the command name and a space are inserted on the command line. Parameters can then be typed in the normal way, or the RETURN key can be used to execute the command. If more than one command matches the letters typed, the first command found in the command table that matches the characters is used.

If the command is not the one desired, the up and down arrow keys may be used to step through the commands in the command table until the desired command is found.

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

Command parameters fall into two groups: required and optional. Required parameters may be entered in the order specified in the command description. If a required parameter is omitted, the command processor or utility will prompt for it. If a RETURN is used immediately, the command is aborted. The required parameter can be followed by any remaining required parameters and optional parameters if desired.

If a name is typed that is not found in the command table, the system attempts to find a file by that name. If one is found, the file is executed. Parameters are passed to the binary program in the same way that they are passed to utilities.

The command processor is case insensitive. This means that commands and their parameters can be entered as either uppercase or lowercase letters. In fact, uppercase and lowercase letters can be mixed.

File Names

In addition to the standard ProDOS names, ORCA allows the use of a physical device number in full path names, following the convention of the Apple /// SOS operating system. For example, if the disk in slot 6, drive 1 is named /ORCA, the following file names are equivalent.

```
/ORCA/MONITOR          .D1/MONITOR
```

The SHOW UNITS command can be used to see what device numbers are assigned to what physical devices; in general, the device numbers are assigned sequentially to all active disks in volume search order.

In addition, two non-disk devices are supported, .PRINTER and .CONSOLE. These are useful when redirecting input and output (see below).

Types of Text Files

ProDOS defines and uses ASCII format files with a TXT designator. ORCA fully supports this file type with its system editor, but requires a language stamp for files that will be assembled or compiled, since the assembler or compiler is selected automatically by the system. As a result, a new ASCII format file is supported by ORCA. This file is physically identical to TXT files; only the file header in the directory has been changed. The first byte of the AUX field in the file header is now used to hold the language number, and the file type is $B0, which is listed as SRC when cataloged from ORCA.

One of the language names supported by ORCA SRC files is TEXT. TEXT files are used as inputs to a text formatter. In addition, PRODOS can be used as if it were an ORCA language name, resulting in a ProDOS TXT file. TXT files are also sent to the formatter if an ASSEMBLE, COMPILE, or FORMAT command is issued.

Input and Output Redirection

ORCA supports full redirection of all input received from the $38 hook and all output sent to the $36 hook. All ORCA system programs that do not use direct positioning of the cursor on the screen send output through the output hook, and all of them except the editor support the input hook, as do most other programs written for the Apple //.

To redirect input, place a < character followed by a valid input device anywhere on the command line. Valid input devices include .CONSOLE and any file name for a ProDOS TXT file or ORCA SRC file. If disk files are used for input, the disk must remain on line until the command finishes. If the file finishes and the program continues to ask for input, it will receive $00.

To redirect output, place a > character followed by a valid output device anywhere after the command. Valid output devices include .CONSOLE, .PRINTER and any valid file name. If a disk file is used as the output device, the file type will match the current system language, and the disk must remain on line until the command finishes.

Both input and output redirection may be used at the same time. Each requires 1K of RAM for the open file buffer, which is taken from the highest available RAM location.

If either input or output redirection is used, all characters involved are removed from the command line before it is sent to utilities or user programs.

Command Files

Command files can be used under ORCA. These are similar to DOS and ProDOS EXEC files under BASIC. A command file consists of a sequence of commands entered into an ORCA SRC file just as they would be entered from the keyboard in response to the command processor prompt. The language for the source file must be EXEC. Input and output redirection are allowed in a command file. Command files may not execute other command files.

To execute a command file, simply type the name of the file. When the system attempts to execute the file, it will discover that it is an EXEC file and use it as a sequence of command processor commands.

A command file removes 1K from usable RAM for its I/O buffer. This memory is taken from the highest available RAM location.

Command and Utility Reference

This section provides a technical description of all of the built in commands and all of the utilities provided with ORCA 4.1. These are listed in alphabetical order for easy reference. Each command's description tells if it is a built in command or a utility. Table 1 lists all of the commands, with an asterisk beside those that are built in.

Following the command name are any parameters for the command. Optional parameters are enclosed in brackets, like [ORG=val]. When one of several choices can be made, this is indicated by separating the choices with a vertical bar. For example, [LIST ON|OFF] indicates that either LIST ON, LIST OFF, or no entry at all can be used with the ASML command.

Keywords which must be typed exactly as shown are shown in uppercase, while descriptions, like "source file" in the ASML command, are shown in lowercase. When the command is entered, either uppercase or lowercase may be used in all places, since the command processor is case insensitive.

*	ASML	Assemble and link
*	ASMLG	Assemble link and go.
*	ASSEMBLE	Assemble.
*	CATALOG	Catalog a subdirectory.
*	CHANGE	Change language of source file.
*	CMPL	Compile and link.
*	CMPLG	Compile, link and go.
*	COMMANDS	Modify command table.
*	COMPILE	Compile, link and go.
*	COMPRESS	Compress and/or alphabetize directories.
*	COPY	Copy files.
*	CREATE	Create new subdirectories.
	CRUNCH	Compress object modules.
	DCOPY	Copy a disk.
*	DELETE	Delete a file.
*	DISABLE	Disable file attributes.
	DISASM	Disassembler.
	EDIT	Edit a source file.
*	ENABLE	Enable file attributes.
*	FILETYPE	Change the type of a file.
*	HELP	List the commands in the command table.
	INIT	Initialize a disk.
*	LINK	Link an object module.
	MACGEN	Macro file generator.
*	NEW	Clear the edit buffer and enter the editor.
	PEEK	Disk zap utility.
*	PREFIX	Change the default prefix.
*	QUIT	Return to ProDOS.
*	RENAME	Change a file name.
*	RUN	Assemble, link and go.
	SCAN	List routines in an object module.
*	SET	Set system characteristics.
*	SHOW	Show system attributes.
*	SWITCH	Change the order of file in a directory.
*	TYPE	Type a source file.
	XREF	Generate a cross-reference.

Table 1 System Commands

ASML file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Assemble a source file. If the maximum error level found is less than or equal to the maximum allowed error level (defaults to 0 unless overridden in the source file) the resulting program is linked. This is a built in command, but control will be passed to one or more languages, and the link editor.

Parameters:

File	The name of the source file to be assembled.
LIST	Indicates whether or not the language will produce a source listing. If omitted, the language will choose its own language dependent default.
SYMBOL	Indicates whether or not the language will produce symbol tables. If omitted, the language will choose its own language dependent default.
ORG	A hexadecimal or decimal number follows, indicating the start location for the finished program. The value only effects the load module produced by this assembly, and can be changed for a later link edit. If the source file contains an ORG directive, its value will override this parameter. If omitted, the default ORG is $2000.
KEEP	Indicates the name of the root file for saving the object modules and the finished program. If this parameter is omitted, the source file must contain a KEEP directive for any output to be saved to disk.
NAMES	A list of subroutine names is enclosed in parentheses. Only the named routines are assembled. Selection of the most recent version of each subroutine is automatic. (See Chapter 13 for details.)

ASMLG file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Assemble a source file. If the maximum error level found is less than or equal to the maximum allowed error level (defaults to 0 unless overridden in the source file) the resulting program is link edited. If the maximum error level is still acceptable, the resulting program is executed. See ASML for a description of the parameters. Like ASML, this is a built in command, but it transfers control to the language translator, link editor, and finished program.

ASSEMBLE file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Assemble a source file. See ASML for a description of parameters. Note that although the ORG parameter can be used, it really has no effect, since the link editor is not called. This is a built in command, but control is passed to the language.

CATALOG [file]

A catalog of the indicated directory is displayed on the screen. If file is omitted, the current prefix is used. The format differs slightly from ProDOS: the fields have been compressed so that all five ProDOS privileges can be displayed, and so that ten character language names can be shown in the auxiliary field. This is a built in command.

CHANGE file language

Changes the language of file from whatever it is to language. This also allows conversions of source files to ProDOS text files (use PRODOS as the language) and ProDOS text files to ORCA source files. This is a built in command.

CMPL file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Same as ASML, but more mnemonic for use with high level languages.

CMPLG file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Same as ASMLG, but more mnemonic for use with high level languages.

COMMANDS file

The system command table is read from file. The command table read replaces any previous commands. The command table can contain blank lines, comment lines, or command definitions. Comment lines start with a semicolon or asterisk. Command lines have four fields separated by blanks. The first field is the name of the command, which must be less than 16 characters and must be a valid ProDOS file name. The second field is one of three characters. If it is a U, the command is treated as a utility, and the third field is not needed. If the second field is a C, the command is treated as a built in command, and the third field is the command number. If the second field is an L, the command is treated as a language name, and the third field is the language number. The last field, in all cases, is an optional comment field.

See the SYSCMND file on the system prefix for an example.

COMPILE file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Same as ASSEMBLE, but more mnemonic for use with high level languages.

COMPRESS [file] A|C|A C

The COMPRESS utility command can alphabetize and crunch subdirectories. When the A parameter is used, the file names in the given subdirectory are placed in alphabetical order. Subsequent uses of CATALOG will show them that way. The C parameter causes the holes in the directory to be moved to the end of the directory. Holes are left in a directory when files are deleted from the middle of a list of files. Subsequent file creations will occur not at the end of the list of files, where they are expected, but in one of the holes. Removing the holes causes the new file name to appear at the end of the list.

COPY file1 file2

File1 is the name of a file to copy, and file2 is the directory or file to copy it to. If file2 is the name of an existing directory, the file is copied to that directory using its old file name. If file2 is not a directory, it is used as the path and file name for the new file. If file2 is omitted, the default directory is used. Wildcards may be used in file1, but not file2. This is a built in command.

CREATE directory

A subdirectory is created, using the name directory. This is a built in command.

CRUNCH file

File must be the base name of an object module. For example, if an assembly and subsequent partial assemblies have produced the object modules MYFILE.A, MYFILE.B and MYFILE.C, then file would be MYFILE. The CRUNCH utility combines all of the object modules produced by the partial assemblies into a single .A file, deleting the subsequent files as it goes. If a segment in a subsequent file duplicates an earlier one, the earlier one is replaced by the new one. New segments that do not replace an existing one are placed at the end of the .A file. Thus, if no new segments have been added, a subsequent link edit will place all of the segments in the same order that they appeared in the original source file.

With this utility, a full assembly will rarely be needed. The cases where a full assembly becomes necessary are:

1. One of the following directives has been changed, and it is not possible to easily identify all of the segments that the change effects.

```
GEQU                    IEEE
MCOPY                   MDROP
MLOAD                   RENAME
65816                   65C02
MSB
```

2. A segment has been deleted or renamed.

3. The disk is full. CRUNCH requires a small amount of work space on the disk. (The amount varies, but is usually only a few blocks.)

CRUNCH is a utility. See also the SCAN utility.

DCOPY dev1 dev2

The disk in dev1 is copied, in its entirety, to the disk in dev2, destroying the contents of that disk. The second disk must be initialized as a ProDOS disk before the command is used. Both volumes must be of the same size. Volume names may be used instead of device numbers, if desired. This is a utility command.

DELETE file

The indicated file is deleted. This is a built in command.

DISABLE D|N|B|W|R file

ProDOS has a total of five privileges which can be separately enabled and disabled for each file. This command allows each of the privileges to be turned off. When a file is cataloged, the enabled privileges are shown using the same letters used here to enable and disable the privileges. As indicated, at least one privilege flag is required, but more than one may be used if the flags are typed with no imbedded blanks. Note that although B can be entered here, it has no effect: only a backup utility is allowed to turn off the backup bit. This is a built in command.

Parameters:

D	Disable delete privileges
N	Disable rename privileges
B	Disable backup required flag (has no actual effect)
W	Disable write privileges
R	Disable read privileges
file	File to change the privileges on.

DISASM

The disassembler is invoked. See Chapter 14 for a complete description of this utility.

EDIT file

The system editor is entered, loading file. This is a utility command. See Chapter 12 for details on the editor functions.

ENABLE D|N|B|W|R file

Allows the ProDOS privilege flags to be set, granting the privilege. See DISABLE for a description of the parameters. Note that ENABLE can turn on the backup bit, even though DISABLE cannot turn it back off. This is a built in command.

FILETYPE file filetype

This command changes the type of a file. Only the type flag is changed. The command must be used with care: changing between some file types can be dangerous, since both ProDOS and ORCA make assumptions about a file and its structure based on its type.

File is the name of the file or files whose type will be changed. Filetype is the type that the file will become. Filetype can be specified as a hexadecimal or decimal number, or as the type characters shown when the files are cataloged.

The most common use of this command is to change a binary file that has been assembled and linked into a system file, so that ProDOS can execute it directly. For example, to change MYFILE to a system file, any of the following commands could be used.

```
FILETYPE MYFILE SYS
FILETYPE MYFILE 255
FILETYPE MYFILE $FF
```

HELP [file]

If file is omitted, this command displays a list of all of the commands in the command table. If file is coded, the system looks for a TXT or SRC file by the given name in a subdirectory called HELP, contained in the utility subdirectory, and prints the file. This gives a powerful, user extendable online help facility. Help files are included for all commands described in this chapter. This is a built in command.

INIT dev1 [name]

The disk in dev1 is initialized as a blank ProDOS disk, and given the name name if it is specified, and BLANK if it is not. If an existing disk is being reformatted, dev1 may be specified as the name of the volume, if desired. INIT supports 35 track floppies and any device that follows Apple's protocall for a disk. This includes the 3.5" floppy disk from Apple, and all hard disks tested to date. INIT is a utility command.

LINK file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

The file name is used as the root file for the object modules. The object modules are linked. See ASML for a description of the parameters. This is a built in command, but control is passed to the link editor. See Chapter 13 for a detailed description of the link editor.

MACGEN [+C|-C] sourcefile outfile macrofile1 ...

An interactive utility scans sourcefile and builds a list of the macros used by the program. COPY and APPEND directives are resolved. When the entire program has been scanned, the utility reads the name of the output file. It then asks for the name of an existing macro library. Wildcards may be used for this response. The macro library is searched, and any needed macros are written to a temporary file on the work prefix. If there are still some unresolved macros, these are listed and the process repeats. Entering a RETURN without a file name indicates to the program that it should stop, even if all of the macros have not been found. When all macros have been found, or RETURN is used to indicate that no more are to be searched, the temporary file is written to outfile and deleted.

The C switch, which defaults to a (+), controls compression of the output file. If on, all excess blanks are removed from the macro to save space. If GEN ON or TRACE ON will be used, it is best to use the -C switch to prevent compression.

Because of the use of a temporary output file, it is possible to specify an old version of outfile as one of the files to be searched.

NEW

Enters the system editor without loading a source file. A file name should be provided when the file is saved by the editor.

PEEK [file]

This utility command executes a disk zap utility which allows individual blocks on a disk to be examined in either hex or ASCII format, changed, and written back to any block on the same device. Since this totally disregards normal file conventions, it can be very dangerous if misused. Once in PEEK, commands are entered as control characters. The commands are:

<- Move the cursor left.

The cursor is moved left one character. If the cursor was on the left-most character of a line, the cursor is moved to the last character of the next line up.

-> Move the cursor right.

The cursor is moved right one character. If the cursor was on the right-most character of the line, it is moved to the start of the next line.

up arrow Move the cursor up.

The cursor is moved up one line. If the cursor was on the top line, it is placed on the bottom line. Use CTRL K on Apple][+ computers.

down arrow Move the cursor down.

The cursor is moved down one line. If the cursor was on the bottom line, it is placed on the top line. Use CTRL J on Apple][+ computers.

CTRL-Q Quit

Exit the utility.

CTRL-R Read a Block

The user is prompted for a block number. It can be entered in decimal, or in hexadecimal if preceded by a $ character. That block is read into memory and displayed on the screen.

CTRL-W Write a Block

The user is prompted for a block number. The block on the screen is written to the indicated block number on disk. Changes made on the screen are not permanent until this command is used.

CTRL-P Print

The block currently displayed on the screen is sent to the printer. The format, (ASCII or Hex) also matches the screen display.

CTRL-S Reverse High Bit

Characters are normally displayed in standard ASCII format. Using this command reverses the high bit of each character before displaying it. The effect is reversed by using another CTRL-S. This has no effect on the hexadecimal display.

ESC Change Display Screen

The type of display is reversed. If the display was in hexadecimal format, characters are shown. If the display is in character format, hexadecimal values are shown.

PREFIX directory

The default prefix is changed to directory, which is the name of a subdirectory or volume. This is a built in command.

QUIT

Control is returned to ProDOS, which prompts for the next system program to be executed. This is a built in command.

RENAME file1 file2

The name of file1 is changed to file2. File1 may use wildcards; if it does, the first file name matched is changed. Wildcards cannot be used in file2. This is a built in command.

RUN source file [LIST ON|OFF] [SYMBOL ON|OFF] [ORG=val] [KEEP=file] [NAMES=(sub[,sub])]

Same as ASMLG.

SCAN file

File must be the full name of an object module. The segments in the object module are listed in the order that they appear, as well as whether each segment is a code segment (produced by a START END pair) or data segment (produced by a DATA END pair). This is a utility command.

SET [variable [value]]

ORCA uses several predeclared variables to control the action of the system. The SET command allows these variables to be changed. The table below lists the variable names, and tells what they effect and what value should look like for each.

If variable is missing, a list of all variables and their values is printed. If variable is coded, but value is missing, the value of that particular variable is listed.

Note that the SET command can be used in the LOGIN file to make changes to the system automatically at boot time.

SET SYSTEM directory

Changes the system prefix to directory. The system prefix is where ORCA looks for the monitor, editor, command table, editor macro file and editor tab file. The default is boot/SYSTEM, where boot is the name of the prefix where ORCA.SYSTEM was executed from.

SET LANGUAGES directory

Changes the languages prefix to directory. The languages prefix is where ORCA looks for the linker, assembler, and compilers. The default is boot/LANGUAGES, where boot is the name of the prefix where ORCA.SYSTEM was executed from.

SET LIBRARIES directory

Changes the libraries prefix to directory. The libraries prefix is where the linker looks for libraries when a program makes references to labels that are not in the program. The default is boot/LIBRARIES, where boot is the name of the prefix where ORCA.SYSTEM was executed from.

SET UTILITIES directory

Changes the utility prefix to directory. The utility prefix is where ORCA looks for utilities. A subdirectory in the utility prefix called HELP is where the help command looks for help files. The default is boot/UTILITIES, where boot is the name of the prefix where ORCA.SYSTEM was executed from.

SET WORK directory

Changes the work prefix to directory. The work prefix is used by ORCA for temporary files. If a RAM disk is available, it should be set there. The default is boot/WORK, where boot is the name of the prefix where ORCA.SYSTEM was executed from.

SET CLICK ON|OFF

If click is turned on, the console driver clicks the speaker whenever a key is pressed. Click defaults to off.

SET WAIT ON|OFF

If wait is turned on, the console driver checks to see if a key has been pressed whenever a carriage return is written. If a key has been pressed, it waits until another key is pressed before returning control to the program that wrote the carriage return. Wait defaults to on.

SET PRINTERSLOT number

The slot where the printer driver will look for the printer is changed to number, which must be in the range 1 to 7. The default slot is 1.

SET PRINTERINIT string

The string sent to the printer to initialize it is changed to string. String may be ommited, in which case nothing is sent to initialize the printer.

Coding "" as the string will set the printer initialization string to the null string (nothing will be sent to the printer). Control characters can be coded in the initialization string by preceding the ASCII character by a ~. For example, the default string of CTRL-I 80N would be coded as

```
~I80N
```

SHOW LANGUAGE|LANGUAGES|PREFIX|TIME|UNITS

Shows current system status information. More than one of the parameters can be entered, if desired, but at least one is required. If multiple parameters are entered, they should be separated by spaces. This is a built in command.

Parameters:

LANGUAGE	Shows the current system default language.
LANGUAGES	Shows a list of all of the languages in the command table, along with their language numbers.
PREFIX	Shows the current system prefix.
TIME	Shows the current time, as reported by the ProDOS time call.
UNITS	Shows the available devices, and for disks the name of the volume currently in that device.

SWITCH file1 file2

This utility changes the position of two files in a directory. File1 may use wildcards, but only the first file matched is used. Wildcards cannot be used in file2. Pathnames may be given, but both files must be in the same subdirectory.

TYPE [+n|-n] file [n1 [n2]]

File is listed on the CRT, starting at line number n1 and continuing to line number n2. If n2 is omitted, the file is listed to its end. If n1 is missing, the entire file is listed. N is a switch. It can be omitted, in which case the file is listed without line numbers. If used, it is normally coded as +N. There must not be a space between the + and the N, and the switch must come after the command, but before the file name. If +N is coded, each line listed begins with a line number. -N can also be coded, but it is the default, so it has no effect. This is a built in command.

XREF [+L] [-L] [+X] [-X] [+F] [-F] [subrange] file

A global cross reference of file is generated. For each language used, the utility expects to find a file by the name of XREF.name, where name is the name of the language, on the utility subdirectory. These files are supplied with the language. XREF is capable of scanning entire programs, resolving COPY and APPEND directives and changing languages. Three sets of switches control the output produced. The switches, as well as the subranges, can be specified in any order, so long as they precede the file name.

The L switch, which defaults to +, is used to specify if the source file should be listed. The listing that is produced has line numbers to the left of each line; it is these line numbers which are referred to in the cross reference. In general, the line numbers correspond to the line numbers produced by the assembler or compiler, but can be different if partial assembly directives are used in an assembly language source file or if more than one language is used. If a listing is not needed, specify -L.

The X switch, which defaults to +, controls the generation of the actual cross reference. Each symbol that appears in the source line and is not an operation code is identified and placed in a cross reference table. For assembly language, if the symbol appeared in column 1, it is taken to be a symbol definition, while appearance anywhere else is assumed to be a use of the symbol. Symbols are listed in alphabetical order, followed by the line numbers of all of the lines where the symbol was defined, enclosed in square brackets. The line numbers of the lines where the symbol was used follows the definition line numbers. To turn off the cross reference, use a -X switch.

The F switch controls the generation of a frequency count, and defaults to off. If turned on with a +F switch, the program lists all operation codes

found in the program, sorted in one column by alphabetical order, and in the other column by frequency of use. In both cases, the operation code is accompanied by a count indicating how many times it was used.

Since symbols found by this program do not have scope, they are not thrown away at the end of each segment. This means that the cross reference utility cannot handle the number of symbols found in large programs. To reduce this problem, up to five subranges can be specified. Subranges are specified as pairs of capitol letters separated by spaces and enclosed in parentheses. Only symbols starting with one of the specified subranges are included in the cross reference and frequency count. For example, if the subrange were specified as (AC SS), then only those symbols that start with A, B, C or S would be included. If no subrange is specified, (AZ) is assumed.

Chapter 12: The Text Editor

Text Entry

Upon entering the text editor, (using the monitor EDIT or NEW commands), the user is presented with a screen which can display twenty-two lines of text. These are the first twenty-two lines of the text file in memory; if there is no file in memory, the screen is blank. The memory area occupied by the text file is called the text edit buffer. The twenty-two lines displayed occupy the text edit window. The text edit window allows examination of any twenty-two contiguous lines in the text edit buffer.

What is seen in the window is exactly what is in that area of the text edit buffer. Text is entered into the text edit buffer as keys are typed. Any keyboard character may be entered as text except for the special function keys (<-, ->, up arrow, down arrow, DELETE, TAB, ESC, CTRL, SHIFT, RETURN, and RESET). Keystroke sequences listed in this chapter also allow entry of the characters [,], {, }, ~, |, \ and ', which are not on a standard Apple][+ keyboard.

The character typed replaces whatever character (if any) was at the cursor position, and the cursor advances one space. There are two cases when the cursor does not advance: either the print stop option has been used in the tab line, or the cursor was in column eighty. In either of these cases, the cursor remains in its old position.

Using more than fifty-seven columns will cause wrap-around when the assembled output is sent to an eighty column printer; for this reason, there is an end of line marker in column fifty-seven. If more than fifty-seven columns are being used, reset the tab line as explained on page 100. This would normally be done only if a printer that will print more than eighty columns on a line were being used.

Special Function Keys

The special function keys have slightly different uses than are found when line-editing in BASIC; they are described below:

<-	Moves the cursor one space to the left. If the cursor was in column one, the key is ignored.

->	Moves the cursor one space to the right. The key is ignored if the cursor started in column eighty or if the cursor was on an end of line marker.
up arrow	Moves the cursor up one line. If the cursor was on the top line of the screen, the screen is scrolled up. If the cursor was on the first line in the file, the command is ignored. This function is also available via the escape mode on Apple][+ keyboards, or by using CTRL-J.
down arrow	Moves the cursor down one line. If the cursor was on the bottom line of the screen, the screen is scrolled down. If the cursor was on the last line of the file, a blank line is added and the cursor is moved down. This function is also available via the escape mode on Apple][+ keyboards, or by using CTRL-K.
DELETE	The character before the cursor is deleted, and the line from the cursor to the end is moved back to fill in the space. If the cursor starts in column 1, the key is ignored. Holding the open-apple key down while typing the DELETE key undeletes the last deleted character. These functions are not available on Apple][+ keyboards.
TAB	The cursor is moved forward to the next tab stop or end of line marker. If the cursor starts in column 80 or on the end of line marker, the key is ignored. CTRL-I will also do a tab.
ESC	The ESC key is used for special editor functions. These are described starting on page 96.
CTRL	The control keys are used for special editor functions. These are described starting on page 92.
SHIFT	The SHIFT key allows entry of upper-register keyboard characters. If the shift key modification has been made, using SHIFT will

give capital letters when in the lowercase entry mode. The shift key on the Apple //e or Apple //c is fully functional.

RETURN — The return key places the cursor at the beginning of the next line. If the cursor started on the bottom line of the page, the text window is scrolled up before the return is issued.

RESET — The RESET key is trapped by the editor, and thus has no real effect. It is possible to lose the last few characters typed when RESET is hit, so it is best not to use RESET in the editor.

User Buffers

The String Buffers

The string buffers are memory reserved for storing the search and replace strings. They are filled using the ESC * and ESC : commands. The * string can then be searched for using CTRL Z and CTRL X, and replaced with the : string using CTRL C and CTRL V.

The Character Buffer

The character buffer is a 256 byte area used to save characters deleted using DELETE. Open-apple DELETE recovers the characters, placing them at the cursor location.

Information Window

The last two lines of the screen are used to display status information. The top line normally displays a ruler with tab marks superimposed - each of the ^ characters represents a current tab setting. When search strings, replace strings, or file names are asked for, this line is used to enter the response.

The next line contains the line number and column number where the cursor is positioned, tells how full the file is by listing the percentage of the available file space currently in use, lists the name of the file being edited, and tells what mode the editor is in. Modes are discussed in detail later.

Control Key Commands

Control characters are immediate-response function keys in the text edit mode. This means that as soon as they are pressed, the function that they are used for is performed. To execute a control key function, first press CTRL, and, while holding it down, press the desired alpha key.

Cursor Movement Keys

CTRL S Tab Left

The cursor moves to the left until a tab stop is found in the tab line, or until it comes to the left edge of the page.

TAB Tab Right

The cursor moves to the right until a tab stop or print stop is found in the tab line, or until the end of the line is reached. TAB is actually the same as CTRL-I, so that can be used as well.

CTRL E End of Line

The cursor is moved to the first space past the last character in the line. If the character in column eighty is not a blank, the cursor is placed in column eighty. This allows easy continuation of a line. If a print stop is coded in the tab line, the cursor will not be placed past the print stop.

CTRL W Start of Line

The cursor is moved to the beginning of the line that it is on.

CTRL T Top Home

The cursor is moved to the first column in the top line of the current display window.

CTRL B Bottom Home

The cursor is moved to the first column in the bottom line of the current display window.

CTRL F First Line

The display window is moved to the first twenty-two lines of the current text edit buffer. The cursor is then placed in column one of line one.

CTRL L Last Line

All blank lines are removed from the end of the current text edit buffer. Next, the display window is moved to the last twenty-one lines in the text edit buffer, placing a blank line at the end of the file (and thus at the bottom of the display window). Finally, the cursor is moved to column one of the bottom line of the display window.

This command is generally used to jump to the end of a file to add new text to a partially completed text file.

String Search Commands

The following commands are used to locate any sequence of characters in a file. The string that is being searched for should have already been entered using the ESC command *, although a chance is given to enter the string if one is not found. If search and replace commands are used, the : string is used as the replace string.

CTRL Z Search Up

A search for the current string is made. The search starts at the character immediately before the character occupied by the cursor, and continues up until the string is found or until the beginning of the file is reached. If the string is found, the display window is moved so that the line with the string is at the top of the page, then the cursor is moved to the beginning of the string. If the string is not found, the message

```
*** String not found ***
```

is printed just below the display window. If there was no string in the string buffer, the string entry routine is entered before the search starts.

CTRL X Search Down

This command is identical to the search up command (CTRL Z) except that the search is conducted from the first character following the cursor to the end of the file.

CTRL C Search and Replace Up

The search and replace sequence begins with the question

```
Auto or Manual (A M Q)?
```

If a response of A is given, a string search up is conducted for the * string. If the string is found, it is replaced with the : string; if it is not found the process ends with the cursor in its original position. After replacing a string, another search is conducted until all occurrences of the string have been found and replaced.

If a response of M is given, each successful search is followed with the question

```
Replace (Y N Q)?
```

The string to be replaced is shown in inverse on the top line of the screen. Typing N continues the search without replacing the string. Typing Y replaces the string, then continues the search. Typing Q terminates the search without replacing the string.

If a search and replace is started with no string in the : buffer, the message

```
Replace with null string (Y N Q)?
```

appears on the prompt line. Replying with N allows entry of the : string, while Y results in a search and delete.

CTRL V Search and Replace Down

This is identical to search and replace up, except that the string search is conducted down in the file.

Miscellaneous Commands

CTRL A Enter/Exit Insert Mode

See page 99 for a description of this mode.

CTRL D Delete to End of Line

This command clears the line occupied by the cursor from the cursors position to the end of the line.

CTRL N Toggle Return Mode

Normally, typing the RETURN key returns the cursor to the beginning of the next line. In anticipation of compilers for block structured languages, this option changes the return function so that RETURN goes to the next line, placing the cursor on the first non-blank character. If the line is blank, the cursor goes under the first non-blank character in the line above. If all of the lines on the screen above the cursor are blank, the cursor goes to the beginning of the line. Typing a CTRL N a second time restores the original handling of the RETURN key.

CTRL O Push Lines

This command is used to write lines out to disk. To select the lines to write, place the cursor on either the first or last line, and type CTRL-O. The message MARK will appear in the status area of the screen. Next, move the cursor to the last line of the list to push, and type CTRL-O again. A prompt for a file name will appear; enter the name of the file to write the lines out to and hit RETURN. Any valid ProDOS path name can be used, or RETURN can be used without entering any file name at all. In that case, the lines are written to the file SYSTEMP on the work prefix. Since the work prefix is generally set to a RAM disk, the output is very fast. Finally, a prompt that allows the copied lines to be deleted appears - answer Y or N. Note that the actual order of the selections is immaterial - it is possible to select the last line of a block of text before selecting the first line.

CTRL P Pop Lines

A prompt for a file name appears; enter the name of any ProDOS text file or ORCA source file. The contents of that file are read in and placed at the cursor location. If RETURN is hit before entering a file name, the editor will read in the contents of SYSTEMP from the work prefix.

CTRL Q Quit

This command exits the text editor, going to an exit menu. Options presented in the exit menu include:

R	Returns control to the editor.
E	Leaves the editor and goes back to the monitor. Changes made to the file are not saved.
S	Saves the file to the same file name used when the editor was entered. This does not result in leaving the editor.
N	Prompts for a new filename, and saves the text in the buffer to that file.
L	Loads a new file from disk without leaving the editor.

Any time a RETURN key is hit in response to a prompt, the command is aborted and control remains at the menu level.

CTRL R Remove Blank Lines

All blank lines, beginning with the line occupied by the cursor and continuing to the first non-blank line, are deleted from the text edit window and the text edit buffer.

Escape Key Commands

When the ESC key is pressed, the message

ESC

appears in the status area of the screen, two lines below the edit window. This indicates that the editor has entered the escape mode. The message remains there until the escape mode is exited by pressing any key that does not correspond to an escape mode command, such as RETURN or ESC.

Escape mode commands are executed as soon as the key corresponding to the command is pressed. After most commands, the editor remains in the escape mode after the command is executed. String entry commands return to the edit mode after the string has been entered.

The Repeat Feature

Escape commands may be repeated by setting a repeat count. This is done by typing a decimal number, followed by any valid ESC command. The command is then executed the indicated number of times, to a maximum of 255.

If a number larger than 255 is entered, or if the ? character is used, the next command will be executed as many times as possible. For commands like inserting a line, which have no theoretical limit on the number of times they can be performed, 256 is used.

Cursor Movement Commands

I or Up Arrow Move Up

Moves the cursor up one line. The screen is scrolled up to accommodate moving the cursor if the cursor was on the top line of the text window. If the cursor was on the first line in the file, the command is ignored.

J or <- Move Left

The cursor is moved left one column. If it started in column one, the command is ignored.

K or -> Move Right

The cursor is moved one character to the right. If it starts in column eighty or on a print stop, the command is ignored.

M or Down Arrow Move Down

Moves the cursor down one line, scrolling the page if necessary. Blank lines are added at the end of the file if they are needed.

Text Edit Window Control

E Scroll Up One Line

Scrolls the current text window page up one line, placing the cursor on the next line up. The cursor remains in the same relative position on the screen that it started at.

C Scroll Down One Line

Scrolls the current text window down one line, placing the cursor on the next line down. The cursor remains in the same relative position on the screen that it started at.

W Scroll Up One Page

Moves the edit window to the twenty-two lines immediately before the lines in the current display window. The cursor is left in the same position in the window that it started.

If the beginning of the file is less than twenty-two lines up, the edit window is moved to the beginning of the file.

X Scroll Down One Page

This is the same as scrolling up one page, except that the page after the one in the window is displayed.

Insert and Delete

B Insert Line

Inserts a blank line at the position of the cursor, moving old lines down to make room for the new one.

Y Delete Line

The line that the cursor is on is deleted, moving the following lines up to fill the space.

H Insert Character

Inserts a space at the position of the cursor, moving the rest of the line to the right to make room. Characters scrolled off of the right side of the line are lost.

G Delete Character

The character under the cursor is deleted, moving the rest of the line to the left to fill the gap. A blank is inserted at the end of the line.

Non-keyboard Characters

Characters that are not on Apple][+ keyboards may be entered in the escape mode using the following translation characters. Although this can also be

done from Apple //e and Apple //c keyboards, it is unnecessary, since those keyboards have keys for all of these characters.

key	result
/	\
-	_
+	~
!	|
(	{
)	}
<	[
>	]
%	@
"	^
'	`

Buffer Commands

***** Enter Search String

This command puts the cursor just below the current display window, following an * prompt. Any string valid in the edit mode may then be entered, finishing with a RETURN. This enters the string into the search string buffer for use by the search commands (see CTRL X and CTRL Z above). If RETURN is entered immediately, any existing search string is cleared from the buffer.

While entering the string, three special characters are recognized. They are the left and right arrows, and RETURN. The left and right arrows move the cursor left and right, erasing or entering the characters they pass into the input buffer. Characters erased from the input buffer using the left arrow remain on the screen, as they do when the monitor is in use. The RETURN key ends the string at the current cursor location. Embedded, leading and trailing blanks are allowed, and are significant.

: Enter Replacement String

This is the same as entering a search string except that the resulting string is used to replace the search string using the search and replace commands.

TAB Stops

A Set, clear tab stops

While in the escape mode, the A key sets a tab stop in the current column if one has not been set, and clears one if it is set.

The Insert Mode

Another entry mode is available in the editor, the insert mode. It is entered by hitting the CTRL-A key. Hitting CTRL-A a second time leaves the insert mode. While in the insert mode, typed characters are inserted at the cursor location, and the line from the cursor to the end of the line is moved forward one character to make room for the new character. Note that characters scrolled off the end of the line are lost.

The other effect of the insert mode is that the RETURN key will split the current line, placing the line from the cursor to the end on a new, blank line, and deleting these characters from the original line.

Editor Macros

Editor macros are available on the Apple //e and Apple //c. This allows the substitution of a single keystroke for up to 128 fixed, predefined keystrokes. The system comes with several predefined macros, but these change so rapidly that they are not listed here. See the system initialization supplement for details.

Entering Macros

The Macro entry mode is available whenever the editor is in use. Begin by typing closed-apple ESC. The first ten of the twenty-six available macros will appear on the screen. Each of the macros is associated with an alphabetic key, shown on the left. The left and right arrows are used to move between the three pages of macros.

To enter a macro, type the letter that it is associated with. The letter must be visible to do this. The line will clear, and a cursor will appear. Any sequence of keyboard characters can now be typed, including calls to other macros. Control characters are displayed in inverse, macro calls as an

inverse B followed by the letter, and open-apple commands as an inverse A followed by the key.

After entering all of the new macros, hit the closed-apple key by itself. An opportunity is given to save this to a disk file - they are placed in the system prefix under the name SYSEMAC, where the editor will find them the next time it starts.

Using Macros

To use a macro, hold down the closed-apple key and type the letter corresponding to the macro.

Defining Tab Stops

In addition to setting the tab stops while in the editor via the ESC-A command, it is possible to set the default tab stops. A file called SYSTABS in the system prefix is searched by the editor as it comes up. If the file exists, it should be a series of lines in a ProDOS TXT or ORCA SRC file. For each active language in the system, the file should contain two lines. The first consists of 80 characters, which define the tab line itself. Zeros are used to indicate the absence of a tab stop, ones to indicate a tab stop, and a two to indicate the end of line marker. An end of line marker is not required.

The second line for the language is either a zero or one, indicating that the RETURN key should always go to the beginning of the line (zero) or that it should go to the first non-blank character in the line (one). For block structured languages like C and Pascal, a one is normally used, while for assembly language a zero is more normal.

To determine which pair of lines to modify for a given language, use the SHOW LANGUAGES command to find out the language number for the appropriate language. The tab and RETURN mode lines in the SYSTABS file are entered sequentially by language number, starting at zero and counting up.

Chapter 13: The Link Editor

Overview

This chapter describes the use and operation of the link editor. The link editor has the job of taking individual subroutines and combining them into a complete program. This usually means relocating certain subroutines and telling each subroutine where other subroutines are located. One significant advantage of a link editor is that if a single subroutine has an error, only that subroutine containing the error need be reassembled, rather than the entire source file. The link editor can then combine the new subroutine with the old ones to produce an executable program.

This scheme results in three distinct kinds of files that the ORCA/M system uses. The first is the source file; these files are created using the system editor. Source files are saved to disk from the editor. They have the file type SRC, which may be seen when the disk is cataloged.

The assembler is used to convert the source file into an object module, which in turn is the kind of file the link editor uses. The object module has a file type of OBJ; the link editor can relocate the code contained within an object module. Note that the link editor does not know or care what source language produced the object module. A Pascal or BASIC compiler designed for use with this system could produce the same type of object module as an assembler.

Output from the link editor is in the form of a binary file with file type BIN. This file is ready to be executed using the ProDOS BRUN command. It may be executed from standard ProDOS or directly from ORCA. It can even be moved back to a DOS 3.3 disk and executed from there.

The link editor is invoked by using any monitor command that does an assembly followed by a link edit, such as ASML. It can also be invoked by using the LINK command with a file name. Parameters for the LINK command will be described shortly.

The Link Edit Process

No matter how the link editor is invoked, the process is very much the same. The link editor is a two pass linker. Pass one begins by locating an

object module on disk and loading the module into memory. Subroutines are assigned final memory locations for the binary load module, and the length of the subroutine is calculated. All global labels defined in the object module are assigned values and placed in a symbol table. The process is then repeated with the next subroutine.

After all subroutines have been processed in the above manner, pass two starts over with the first subroutine. This time, global labels referenced by the subroutine are looked up in the symbol table and resolved in the output file. The subroutine is then placed in the load module on disk. This process is then repeated for each of the other subroutines.

Object Modules Created by the Assembler

When the assembler is directed to save the results of an assembly on disk using the KEEP parameter, it was necessary to supply a file name. The assembler then created two disk files. The first file contained the object module for the first assembled subroutine in the source file. (This is the entry point for the finished program.) This first object module was saved with the file name supplied using the KEEP directive, with the suffix .ROOT added to the end. For example, if the file name OBJECT was used, the first subroutine would be saved in a file called OBJECT.ROOT. The remaining subroutines are placed in a file called OBJECT.A. They are placed in this file in the order that they occurred in the source file.

After assembling the complete program, there may be a need to reassemble a few of the subroutines, using the NAMES=(n1 ... nx) assembler option. When this is done, the assembler searches the output disk for an old file with the same root name. If this was the second assembly, it would find the file called OBJECT.A. The newly assembled (or re-assembled) subroutines are saved in a file called OBJECT.B. They appear in the order in which they were encountered in the source file. Subroutines which were not re-assembled are not placed in the new file. The next partial assembly is stored in a file with the name OBJECT.C, and so on.

If the first subroutine is reassembled, it is placed in a separate file called OBJECT.ROOT, replacing the first file by that name.

Subroutine Selection

When a link edit starts, the same KEEP file name as used by the assembler must be provided. (This is done automatically by ASML and similar

commands.) In the above example, this was OBJECT. The link editor scans the output disk for a file with the name OBJECT.ROOT, using the subroutine in that file as the first subroutine in the final binary load module. It then locates the last object module assembled by finding the file with the highest alphabetical suffix. (It does this by scanning successively for files with ascending alphabetic suffixes.) In the example above, this was OBJECT.C. Subroutines are taken from this file in the order encountered, linked and then placed in the load module. The link editor then proceeds to the previous file - that is, the one with preceding alphabetical suffix. If a subroutine is found which has not yet been linked, it is placed in the load module. If the subroutine has already been linked, having been found in a previous (hence more recently assembled) file, it is ignored. Thus, the most recent version of a subroutine is selected automatically.

If there are still unresolved external references, the link editor assumes that these are to be resolved from library files. Library files have the same format as other object modules, but they are located in the library directory. (See the initialization addendum for more on this directory, and how to change it.) The link editor searches the library prefix for library files. If there there are no library files to be found, the linker assumes that the unresolved references are errors.

When the library directory is found, each library file is searched once, in the order in which it appears in the catalog. If any subroutine has a name corresponding to an unresolved reference, it is placed in the load module. A subroutine selected in this manner can have its own unresolved references, which are then resolved during the rest of the library search.

Having found all of the subroutines it can, the link editor proceeds to pass two of the link edit. Pass two produces a BIN type output file with the keep file name as its file name. If there are no errors, the program is ready to be executed. The binary file can be executed directly from ORCA by simply typing the name of the file.

Link Edit Command Parameters

Several link edit options are available. These are entered as parameters to the LINK command. They may appear in any order, and may be separated by commas or spaces. Remember that a file name must be used with the LINK command; the file name comes after the LINK command, followed by the parameters. The following are valid commands:

```
LINK MYFILE,KEEP=MYFILE,ORG=$8000
LINK /ORCA/MYFILE ERROR OFF LIST OFF KEEP=/PROFILE/SAVE
```

KEEP=file

The binary output file (load module) is saved on disk using the file name file. This file can then be executed from either ORCA or Apple ProDOS.

ORG=org

The finished program will start at location org. The value of org may be coded in decimal or hexadecimal. If hexadecimal is used, the value must begin with a $ character. The default is $2000.

LIST OFF

Normal output from the link editor is suppressed. Only errors are listed.

SYMBOL OFF

The global symbol table which is normally listed at the end of the link edit output is suppressed.

Creating Library Files

Several library subroutines are included with the assembler, ready to be used automatically by the link editor. New library files can also be created. Begin by writing a source file as if the library subroutines were part of a main program. A dummy subroutine is placed at the start of the file. This first subroutine will be placed by the assembler into the root file (the one that ends in .ROOT). Assemble the file normally, as if it were a program - there is no need to do a link edit. The root file is simply deleted, and the secondary file (which ends in .A) may be renamed as desired, then placed in the library subdirectory.

All subroutine libraries needed for a given link edit must be on the library directory for the link editor to find them. Libraries are searched one time, sequentially, so if an library subroutine requires another library subroutine, it should precede it in the source file.

Output

In addition to generating the load module, the link editor produces printed output, showing exactly what it did. This output can be suppressed by using LIST OFF, except for errors and the global symbol table. The global symbol table is suppressed by using SYMBOL OFF.

Pass One Output

All link editor error messages list the name of the program segment where the error appeared, as well as how far into the program segment the error was. Errors are the only printed output from pass one.

Pass Two Output

Pass two prints errors the same way as pass one. In addition, pass two writes the starting location, length, and name of each program segment. Each segment is also flagged as Code (produced by a START-END sequence) or Data (produced by a DATA-END sequence).

Global Symbol Table

The listing concludes with an alphabetized global symbol table. Each symbol is followed by the value assigned to it, given in hexadecimal.

This value is followed by a single hexadecimal digit. The digit is zero if the label is a subroutine name or global label. If the label was defined in a data area, the digit indicates the number of the data area that defined it.

File Length and Error Count

After the global symbol table has been printed, the link editor prints a message stating the number of errors (if any), and what the highest error level was. (Error levels are explained in Appendix A.) This message is not printed if there were no errors. The last line tells where the program starts and how many bytes long it is, in hexadecimal.

Chapter 14: 6502/65C02 Disassembler

Introduction

A disassembler is a utility for re-constructing a version of the source code for a program from a machine code version of that program. The disassembler accepts a machine language program in the form of a binary file as input, and generates assembly-language source code for that program, in the form of a SRC file, as output.

Reasons for using or needing a disassembler range from a desire to understand or modify a program which is only available as object code; re-constructing a program for which the source code has been lost; or even as a means to convert an existing program from one source format to another - in this case, the ORCA source format, as naturally that is what this disassembler produces.

The disassembler will try and automatically make reasonable guesses as to which bytes in the raw binary file represent code, and which are data, as well as provide simplistic labels for all memory locations referenced in the operand field of instructions, but it cannot be expected to do a very good job on its own. Accurate determination of just what is code and what is data, the appropriate format for the data, and meaningful names for labels, is up to the operator of the disassembler.

This disassembler is therefore an interactive, command oriented program. It uses a command interpreter similar to the one in the ORCA monitor. To enter the disassembler, type DISASM from the ORCA/M monitor.

Just as the pound sign, #, is the command-line prompt for the ORCA monitor, the disassembler likewise has a command prompt. It is the colon; whenever the colon appears, followed by a flashing cursor, any of the disassembler commands listed below may be entered. As with the monitor, when commands are entered in response to the command line prompt, a blank must separate commands from operands (such as filenames or addresses). In contrast to the monitor, the full command name does not need to be typed. Instead, only enough characters of a command need be entered to distinguish it from those preceding it in an alphabetical listing of available commands.

The actual technique of disassembling a program is an art that must be developed by the individual operator. The procedure of creating the disassembled listing is a process of building a symbol table in memory containing all of the labels, DC statements, START and END directives, and other information, that has been entered by the user. As labels are added, and the program re-listed, the disassembled program becomes more and more intelligible as the user-entered labels are interlisted with the remaining, incompletely disassembled code. With each iteration of the process, the symbol table is built up into what becomes the "key" to the entire program, from which the disassembler (using the GENERATE command) can create an actual source listing of the program, which can then be re-assembled to re-create an identical binary version of the program, and modified as desired.

Note that in this process the disassembler can use up large amounts of memory. The more complex and sophisticated the disassembly, the more symbol table space will be required. If a point is reached where an out-of-memory situation occurs, divide the binary machine code file into pieces.

In reviewing the command descriptions, the first and most important to learn is the HELP command. It is provided to give a brief listing of the available commands. Once a basic familiarity with the operation of the disassembler is achieved, it is likely that the HELP screen will be the only reference needed for operation of the disassembler.

Command Descriptions

In the operands for the commands, label means an initial alphabetic character, followed by from zero to nine alpha-numeric characters. Adr means a hexadecimal number in the range $0000-$FFFF. These should not be preceded by the $ character when entered in a command line. Filename indicates a disk file name.

Many commands use two addresses, separated by a period. For a large number of these, both the period and the second address is optional. If only a single address is given, the operation is performed only on the single byte referenced at that address. If the first address is less than the starting address of the program being disassembled, the starting address of the program is assumed. If the first address is followed by a period, but not by the second address, the operation is performed on all locations from the first address to the last address in the program. Using both addresses performs the operation on the range of memory from the first address to the last address, inclusive.

CATALOG [file]

Catalogs the disk. File is the name of the directory to catalog. If omitted, the current directory is cataloged.

CODE adr[.[adr]]

If the memory locations in the range indicated have previously been tagged as a part of a DC statement, they are restored to the default condition, where all binary data is listed as code (program instructions and their operands), rather than as program data.

DATA adr[,label]

A DATA directive is placed at adr in the output file. The disassembler will automatically precede it with an END directive unless it is the first line of the program. If label is coded, it will be placed in the label field of the directive.

DC adr[.[adr]][,type]]

A DC directive is defined, indicating that the memory is to be considered as data, and not as instructions, which is the default case. Type is the type of DC statement desired, and defaults to H if omitted. Valid types are A (address), B (binary), C (character), H (hexadecimal) and I (integer). If a character DC type is requested, the high-order bit of each character will be cleared, regardless of its previous setting. This is consistent with the ORCA/M default of MSB OFF. Non-printing ASCII characters are replaced with the underline character.

DELETE filename

The indicated file is deleted from the disk.

DELLABEL [adr][,label]

The label defined for the given address (or label name) is deleted from the symbol table. The command has no effect if the given label has not been defined.

Either adr or label must be provided. If both are entered, adr is used with a comma.

GENERATE filename

The program is listed (disassembled), and the output is sent to a source file on disk with the name filename.

The generate process starts with the creation of those labels which are still undefined by the user, but which are needed to allow the operand of every instruction to be a label rather than a constant hexadecimal address. Each label so defined by the disassembler is simply an "L", followed by the four byte hexadecimal address of the memory location being referenced in the program. All instruction operands (except immediate operand instructions) and address type DC directives which reference the address will use the label. If an address is referenced which does not begin on an instruction boundary, no label is generated, and the operand is left as a hexadecimal constant.

If the resulting source file is too long to be edited, use the TYPE command to break it into smaller files. To do this, specify the range of lines that will appear in the new, smaller file, and redirect the output of the command to the file.

HELP

This command accesses a built-in listing of all of the commands available.

HEX adr[.[adr]]

The memory locations indicated are listed on the screen. Each line begins with the address of the first byte listed, followed by a colon. Up to sixteen bytes are then listed as hexadecimal numbers. Following the hexadecimal listing, the character equivalents of the hexadecimal values are printed, surrounded on the left and right by periods. If the value does not correspond to a printable ASCII character, it is printed as the underline character.

This command can also be entered as X adr[.[adr]]. This alternate form is provided to allow a single-keystroke version of this frequently used command.

LABEL adr,label

The label given is entered into the symbol table with the value adr. Any time the line starting at adr is listed using the LIST command, or generated to an output file using the GENERATE command, label is placed in the label field of the line. If a listed instruction operand references adr, label is used instead of the address.

If adr is outside the range of the program being disassembled, the label will be placed at the beginning of the program as a GEQU directive. Global equates are listed any time the first line of the program is listed.

LIST adr[.[adr]]

The indicated range of memory is disassembled and printed on the screen. All lines are listed exactly as they will be formed when the GENERATE command is used to send the file to disk, except that the L-type labels created by the GENERATE command are not used.

LOAD filename

A binary file is loaded from disk. The file may then be disassembled using the commands described. Note that all commands with addresses as operands will only operate on the range of memory encompassed by the program which has been loaded. The only exception is LABEL, which will allow a label to be defined outside the range of the program.

The program is actually always loaded into memory at the same fixed location, regardless of the file's actual execution address. Whenever the file is listed, however, the disassembler lists the program as if the origin or initial offset were the execution address of the file. Loading a program resets the symbol table pointers, deleting anything that was in the symbol table.

PARAMETERS

The real starting location (execution address) and length of the program currently in memory (the last one loaded using the LOAD command) is listed on the screen.

PROFF

Output is no longer sent to the printer.

PRON

Anything that is normally printed on the screen is routed to the printer.

QUIT

Control is returned to the ORCA/M monitor.

START adr[,label]

A START directive is defined at the location specified by adr. Unless it is the first line of the program, the disassembler automatically precedes it by an END directive in output listing, as well as a comment indicating the original location of the directive in the program. If label is entered, it will be used in the label field of the START directive. The label can also be added later using the LABEL command, or may have already been defined.

TLOAD filename

A symbol table is loaded from disk, replacing the currently resident one. This allows the disassembly process to be resumed after interruption.

TSAVE filename

The current symbol table is saved on disk under the file name filename. The symbol table contains all of the definitions entered by the user in creating the disassembly listing. By saving the symbol table, a session may be interrupted, and continued at a later time by re-loading the machine code binary file using LOAD, and the symbol table, using TLOAD. With these two files loaded, the interactive disassembly process can resume as before.

Some Final Comments

It is not intended that the disassembler will produce a completed program, ready to be assembled and executed. In fact, except for very short programs, one will always have to at least split the source file up and add APPEND directives at the end of each file. Disassembly is the first step along the course to a completed source listing. Much time adding comments, other directives, and more descriptive operands will need to be spent before a source listing is truly complete.

Two problems are likely to occur frequently, so they will be mentioned here. The first involves character constant formats. The ORCA system assumes that all characters have the high bit clear, which is how they are defined in the ASCII character set. Some programs will represent characters with the high bit set. This can be easily adjusted by placing the MSB ON directive at the beginning of the source file. Some assemblers also provide a special directive which sets the high bit of the last character string. The best way to deal with this is, having disassembled the entire string as C type DC statements, to then replace the DC directive with a macro to do the bit-manipulation on the characters in the string.

The other problem is that not all assemblers provide a mechanism for defining local labels, which can lead to a style of program referred to as "spaghetti code." For the ORCA assembler, this results in a situation where relative branches take place outside the bounds of logical subroutines. There are two ways to deal with this. The first is to change the program slightly, so that relative branches are made only within the subroutine that they are contained in. This alternative will probably make the program a little longer. The other method is to break up the program wherever possible, ignoring logical subroutine boundaries. ENTRY directives can then be used to allow access to the subroutine at their logical entry point. If this alternative is chosen, be sure and do a full assembly each time the program is assembled, since the subroutines will be very position dependent.

Chapter 15: Running the Assembler

Introduction

The assembler is the heart of the ORCA/M assembly language development system. It is invoked from the monitor by using any of the assemble or compile commands. It then assembles the source file named in the parameter list of the assemble command.

The assembly is not limited to the first source file. For large programs, that file is simply the first of many source files. The file in memory chains to or includes other source files using APPEND and COPY assembler directives. The needed source files are brought into memory automatically. After the assembly is finished, the original source file is loaded back into the text edit buffer.

If macros are used, one or more macro files will be needed by the assembler. The MCOPY and MLOAD directives are used to tell the assembler which macro files to use. When an operation code is encountered in a source file which does not match any instruction or assembler directive, the macro files are scanned for a macro definition. The macro, if found, is then expanded into the source stream and assembled into an output file. Both the source file itself and macro file remain unchanged.

Although the assembler itself, and the initial source file, must be on line when the assemble command is issued, additional source files accessed via COPY or APPEND directives do not need to be in a disk drive when the assembly starts, provided that a full path name is used with the COPY or APPEND directive. Whenever the assembler encounters a full path name, it stops and checks to make sure that the needed volume is online. If not, it performs some house keeping and asks for the volume, then continues the assembly after finding the correct volume. The prompt will look something like

```
Place /ORCA on line.  Press any key when ready.
```

After inserting /ORCA, press any key but ESC, and the assembler will continue. If a program error results in asking for a disk that does not exist, the ESC key can be pressed so that the assembler knows the disk cannot be made available.

Assembler Parameters

All commands which cause an assembly or compile can be followed with a number of parameters which set the assembler options. (Assemble and compile are equivalent. The operating system loads the assembler or compiler based on the language number contained in the file header.) Abbreviation of parameter names is not allowed.

All of the parameters except NAMES have corresponding assembler directives. If both a parameter and a directive are used, the parameter setting is superseded by the assembler directive when the assembler directive is found in the source file. Source lines before the assembler directive is found will be under the control of the command line parameter setting.

KEEP=outfile

Keep File Name

This parameter tells the assembler to keep the object module it produces; that is, store the generated object code to disk. The disk file outfile is used for the output file; the file name must follow ProDOS conventions.

LIST OFF Don't List Output

The normal output from the assembler is suppressed. Only the pass messages, errors and symbol tables will be listed. LIST ON can be coded, but it is redundant, since the assembler normally lists output.

NAMES=(n1,n2, ... ,nx) Subroutines to Assemble

The assembler will assemble only the subroutines whose names are listed. Subroutines are given names using the START directive, as explained on page 149. The output object file will contain only the subroutines listed; however, it will be named as part of a collection of files from previous assemblies of the same source file in a way that allows the link editor to link files from different assemblies together. See the operating system reference manual for a description of how this is done.

Assembler directives which are global in scope will still be resolved, whether or not they are in one of the subroutines assembled. These directives are:

ABSADDR	generate absolute addresses
APPEND	append a source file
COPY	copy a source file
ERR	print errors
EXPAND	expand DC statements
DC	declare constant
GEN	generate macro expansions
GEQU	define global symbolic constant
INSTIME	generate instruction times
IEEE	enable IEEE format numbers
KEEP	keep output file
LIST	list output
MCOPY	copy macro file
MDROP	drop macro file
MERR	maximum error level
MLOAD	load macro file
MSB	set most significant bit
PRINTER	send output to the printer
RENAME	rename op code
SYMBOL	print symbol tables
65816	enable 65816 op codes
65C02	enable 65C02 op codes

The operands of these directives cannot contain labels unless they appear inside a program segment, and the segment that they appear in is assembled. If these rules are not followed, an invalid operand error will result. The directives themselves are described in Chapter 17.

ORG=org Set Origin

Assembly will begin at the fixed memory location specified to the right of the equal sign. The origin may be coded in decimal or hexadecimal. If hexadecimal numbers are used, the number must be preceded with a $ character.

SYMBOL OFF Symbol Table Off

Symbol tables are not listed. SYMBOL ON can be coded, but is redundant, since the assembler prints symbol tables by default.

The Assembly Process

Pass One

The source file is assembled one subroutine (program segment) at a time. Each subroutine goes through two passes. The first pass resolves local labels. When pass one encounters an END directive or the end of the source file, it passes control to pass two. Lines which appear outside of program segments do not contain labels, so they can be completely resolved in pass one.

Pass Two

When pass two is called, it starts at the beginning of the program segment, as defined by the START or DATA directive. Pass two then assembles each line for the last time. Pass one has already resolved any local labels, so pass two can produce both the object code output and the assembly listing. External labels are resolved as $8000, possibly with some offset value. External zero page labels, indicated in the source listing by a < character before the expression, are resolved to $80.

When pass two finishes with a subroutine, it prints the local symbol table. It then passes control back to pass one to begin the next subroutine. If there are no more subroutines to assemble, control is returned to the operating system. Depending on the "assemble" command given, the operating system passes control to either the monitor or link editor. If the link editor is called, it uses the object modules created by the assembler as input. These are relocated and global labels are resolved, giving an executable binary file as output.

Controlling the Speed

This assembler has a throttle: paddle zero can be used to slow down the assembly listing as pass two prints it. This is desirable when only screen output is being produced, rather than using the PRINTER ON directive to create an assembly listing on a printer. To slow down the listing, turn paddle zero counterclockwise. Speed can be recovered by turning the paddle clockwise.

Stopping the Listing

At any time during pass two, the assembly may be stopped by pressing any keyboard character. Note that the assembly will stop only if a line or symbol table is being printed, and not for the pass headings (which lists the subroutine name). This provides a quick way to scan for errors; by turning off the listing and symbol table, only the output of error lines can stop the listing. Pressing a key at the beginning of the assembly will then stop the listing at the next error. Since the pass headings are still displayed for each subroutine, the subroutine which contains the error may be determined.

To restart the listing, any key but ESC may be pressed. The listing will continue until another key is pressed to stop it again. If the listing has been stopped, and ESC is pressed, the text editor is entered. The line that would have been printed next will be at the top of the edit page, with the cursor at the beginning of that line.

Terminal Errors

If the assembler encounters a terminal error (such as a symbol table overflow), it returns control to the operating system. The operating system then enters the text editor automatically, and places the line that caused the error at the top of the text edit window. This allows identification of the offending line, even if pass two had not started and no listing had been produced yet.

A list of terminal errors is contained in Appendix A.

The RESET Key

If the RESET key is pressed during an assembly, control is returned to the operating system. The operating system enters the text editor as if a terminal error has occurred. The editor is entered, and the current line is displayed, showing where the assembly process had been interrupted. If this was done during a macro resolution, the line displayed is the macro call statement. If the RESET key is used while a disk write is in progress, information may be lost on the diskette.

The Assembly Listing

Screen Listings

A listing on the screen is produced during pass two unless the assembler is instructed not to list the output. Each subroutine begins with two messages announcing the subroutine name and pass. The listing continues with the assembled code.

Each output line has four parts. The first part is a line number. This is a four digit decimal number, starting at 0001 on the first line and incrementing for each source line. The line number is incremented even if the output line is not listed. Thus, even if listing is turned off for part of the assembly, it is still possible to know exactly how many lines the assembler has processed. Lines generated by a macro are not considered source lines, so they do not have a line number.

Next is the current relative address. This is the memory location that the code would be at if the subroutine were placed at location $0000 by the link editor. (Despite this offset, labels defined relative to the program counter within the range zero to $FF are not zero page; the origin of $0000 is simply for convenience in calculation. Internally, the actual origin in the relocatable object module is $1000. The link editor outputs indicate where the subroutine is actually located in a given binary file.) Next comes a sequence of up to four bytes, printed in hexadecimal. This is the code that was generated by the assembler. Finally, the source statement that generated the code is printed.

If an error is detected in the source statement, it is printed on the next line. All error messages are text messages, not simply error numbers. The errors are explained in Appendix A.

Printer Listings

If the PRINTER ON directive is issued in a source file, subsequent lines are sent to the printer. The assembler expects an eighty column printer with an interface card in slot one.

Printed listings are generally the same as listings to the screen, except that the messages announcing the start of various passes are not printed. The assembler assumes sixty-six lines per page, and prints on sixty of those lines. Six lines are skipped after each block of sixty lines to allow for page

breaks. After printing the symbol tables for a subroutine, the assembler skips to the top of the next page.

Chapter 16: Coding Instructions

Types of Source Statements

There are four types of lines in an assembly language source listing. The first is the comment line. Its purpose is to allow text to be inserted in the source listing in order to document the program. Two other line types are instructions and assembler directives. They are coded in the same way, and are described together here. The last is the macro call statement, detailed in Chapter 18.

Assembler source file lines may be up to eighty columns long, numbered from one to eighty. Since most printers use eighty columns, assembler source lines should generally be restricted to fifty-seven columns, as twenty-three columns must be allowed for information printed by the assembler. If this is not done, printed assembler output will wrap around to the next line. Aside from making the listing difficult to read, it also causes the assembler to miscount the number of lines of printed output, misplacing future page breaks.

Note that many printers allow compression of text, so that more than 80 columns can be printed on a single line. If at least 103 characters can be printed on a line, wrap-around will not occur.

Comment Lines

There are five forms of lines which are regarded as comment lines by the assembler. They are described by use.

The Blank Line

Any blank line is treated as a comment line. Blank lines are often used to logically separate sections of code.

The Characters *, ;, and !

Any line with an asterisk (*), semicolon (;), or exclamation mark (!) in column one is treated as a comment. Any text in the line is ignored. It will

be printed when the source listing is generated by the assembler. Note that symbolic parameters are expanded whether or not they are in a comment.

The Period

Any line with a period (.) in column one is treated as a comment. These lines are not printed in the source listings produced by the assembler, unless the TRACE ON directive has been used. These lines are intended for use as labels for conditional assembly branches. (See AIF and AGO in Chapter 18.)

Instructions

An assembly language statement, whether an instruction, directive or macro, has four basic parts. The only exception to this format is a line that contains only a comment, as discussed above. These four fields are the label, operation code, operand and comment.

The Label

Each line may begin with a label, which is required for a few directives. The label must begin in column 1, and cannot contain imbedded blanks. Each label starts with an alphabetic character, the tilde (~), or the underscore (_), and is followed by zero or more alphanumeric characters, tildes (~) or underscores (_). Both tildes and underscores are significant. Labels may be any length, but only the first ten characters are significant.

Note that labels starting with the tilde character are reserved for use in macros and libraries supplied by the Byte Works.

It is best not to use A as a label, since it can cause confusion between absolute addressing using the label A and accumulator addressing.

The Operation Code

The operation code field is reserved for an assembly language instruction, assembly directive, or macro. At least one space must be left between the label and the operation code. If no label is coded, the operation code can begin in any column from two to forty. Normally, the operation code begins in column ten. The tab line has a tab stop in this column for convenient placement.

Operation code mnemonics for machine-language instructions are always three character alphabetic strings. The assembler allows the following substitutions for the standard 6502 operation codes:

Standard	Also Allowed
BCC	BLT
BCS	BGE
CMP	CPA

Assembler directives vary in length from two to seven characters. The operation codes for assembler directives are listed in Chapter 17 and Chapter 18. Macro operation codes, which are a form of user-defined operation code, are described in Chapter 18.

The Operand Field

The operand is the information that the instruction uses to perform its function. There must be at least one space between the operation code and operand. The operand normally starts in column sixteen; a tab stop is provided to allow easy movement to that location. Formats for the operand field vary a great deal. Refer to the descriptions of the individual operation codes for the format to be used in forming their operand fields.

Instruction Operand Format

Assembly language instruction operands all consist of basically two parts: a number and a few characters that indicate the kind of addressing mode. For example, 400 is a valid operand. It is treated as an absolute address by the assembler. With the addition of two characters one gets 400,X, which is a different addressing mode called absolute indexed. However, the number is still the same.

In ORCA, the number can take on many forms. These forms are covered in detail in the next section, when expression syntax is covered. For now, only one aspect of the expression is important, and that is whether that expression is a constant or whether it involves external references. If all of the terms in an expression are constants, i.e, they are numbers or labels whose value are set by EQU directives or GEQU directives, then the assembler can determine the final value of the expression without the aid of the link editor. In that case, the expression is a constant expression. If any

term in the expression is a label that must be relocated, the expression itself must also be relocated. This distinction is important, since the assembler is able to automatically select between addressing modes that offer one, two, and three byte variations only if the expression is a constant expression. In the case of a relocatable expression, the assembler will always opt for the two byte form of the address, unless it is explicitly overridden. The length of addressing used can be forced by using a < before the expression to force zero page addressing (called direct page addressing on the 65816), a | to force absolute addressing, and a > to force long addressing. This is illustrated in the operand format table, below. Note that long addressing is only available on the 65816. Also note that a ! character can be used instead of |, just in case the keyboard does not support |.

Operands for immediate addressing are resolved to one, or on the 65816 occasionally two, bytes. It is necessary to be able to select which byte or bytes to use from an expression. Three operators are provided to select the appropriate bytes from the value. These operators must appear immediately after the # character, which indicates immediate addressing. If no operator is used, the least significant byte (or bytes) is used. This also happens if the < operator is used. The > operator has the effect of dividing the expression value by 256, selecting the next most significant byte. Finally, the ^ operator divides the expression by 65536, moving the bank byte into the least significant byte position.

The following table shows all legal operands of both the 6502 and 65816. The labels ZP, ABS and LONG refer to constant expressions that resolve to one, two or three byte values, respectively. EXT is a relocatable expression.

Addressing Mode	Operand Format
Implied	none needed
Immediate	#ZP
	#>ABS
	#<ABS
	#^ABS
	#ABS
	#>LONG
	#<LONG
	#^LONG
	#LONG
	/ABS

	/LONG
Zero Page (Direct Page)	ZP
	<EXT
Absolute	|ZP
	ABS
	EXT
Absolute Long	>ZP
	>ABS
	LONG
	>EXT
Relative	ABS
	EXT
Zero Page Indexed	ZP,X
(Direct Page Indexed)	ZP,Y
	<EXT,X
	<EXT,Y
Absolute Indexed	|ZP,X
	|ZP,Y
	ABS,X
	ABX,Y
	EXT,X
	EXT,Y
Absolute Long Indexed	>ZP,X
	>ABS,X
	>EXT,X
	LONG,X
Absolute Indirect	(ABS)
Zero Page Indirect	(ZP)
	(<EXT)
Zero Page Indirect Long	[ZP]
	[<ZP]
Zero Page Indirect Indexed	(ZP),Y
	(<EXT),Y
Zero Page Indirect	[ZP],Y
Indexed Long	[<EXT],Y
Zero Page Indexed	(ZP,X)
Indirect	(<EXT,X)
Absolute Indexed Indirect	(ZP,X)
	(ABS,X)
	(EXT,X)
Stack Relative	ZP,S
	<EXT,S
Stack Relative Indirect	(ZP,S),Y

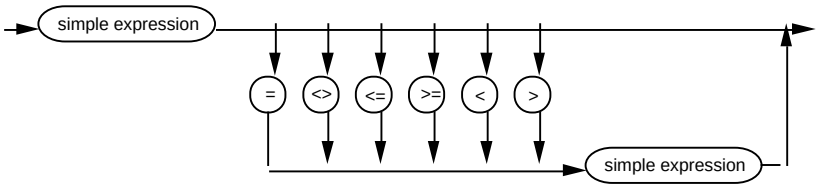
Indexed	(<EXT,S),Y
Accumulator	A
Block Move	EXT,EXT
	ZP,ZP
	ABS,ABS
	LONG,LONG

Expressions

Whenever a number is allowed in an operand field, whether in a 6502 instruction or in a directive, an expression may be used. In their most general form, expressions resolve to an integer in the range -2147483648 to 2147483647. The result of a logical operation is always 0 or 1, corresponding to false and true. If an arithmetic value is used in an assembler directive which expects a boolean result, 0 is treated as false, and any other value is treated as true.

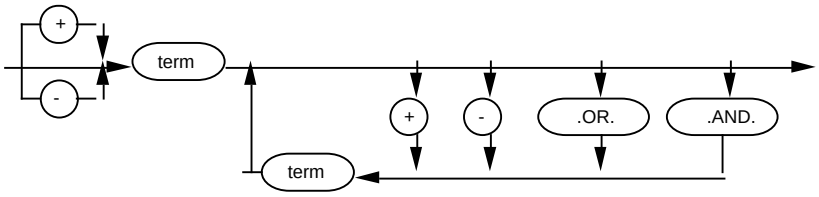
Syntactically, an expression is a simple expression, or two simple expressions separated by a logical comparison operator.

expression



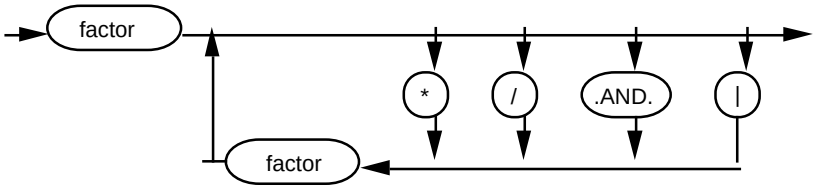
Thus, logical comparisons have the lowest priority. A simple expression is the customary arithmetic expression. Syntactically, this is expressed as an optional leading sign, a term, and, optionally, a +, -, .OR., or .EOR. followed by another term.

simple expression



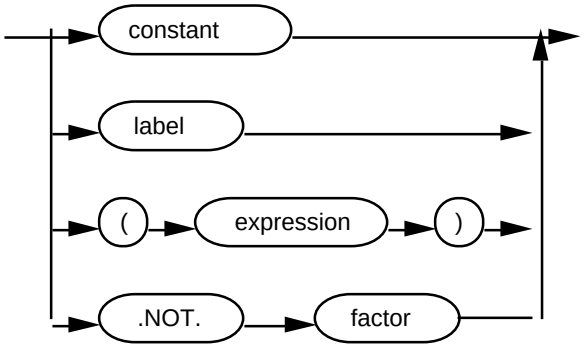
A term is a factor, optionally followed by one of the operators *, /, .AND., or | (the bit shift operator) and another term. .AND. is a logical operator, asking if the terms on either side are true. If both are true, so is the result, otherwise the result is false. The vertical bar (or, optionally, !) is a bit shift operator. The first operand is shifted the number of bits specified by the right operand, with positive shifts shifting left and negative shifts shifting right. Thus, a|b is the same as a*(2^b).

term

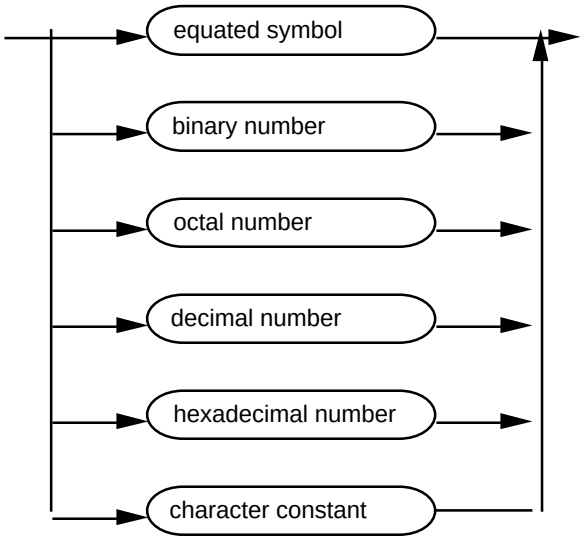


A factor is a constant, label, expression enclosed in parentheses, or a factor preceded by .NOT.. .NOT. is the boolean negation, producing true (1) if the following factor is false, and false (0) if it is true. Here, a label refers to a named symbol which cannot be resolved at assembly time. Constants are named symbols defined by a local EQU directive or global GEQU directive, or a decimal, binary, octal or hexadecimal number, or a character constant.

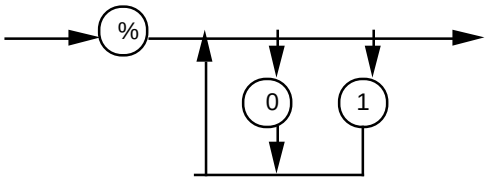
factor



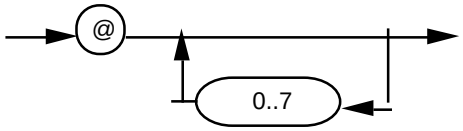
constant



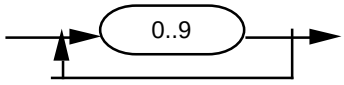
binary number



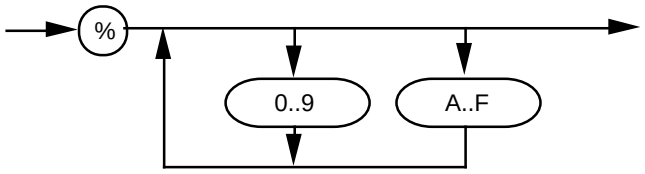
octal number



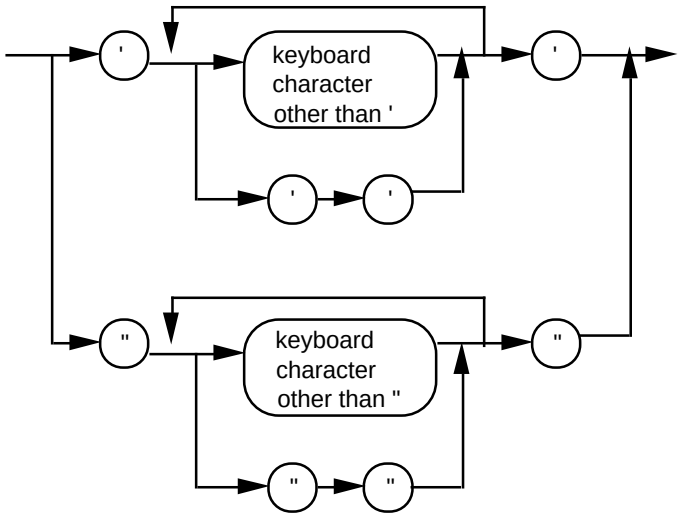
decimal number



hexadecimal number



character constant



The Comment Field

In-line comments can start in any column past the first space after the operand field. If an instruction does not require an operand field, they can start in any column after the first space past the operation. Comments generally start in column forty-one. A tab stop is provided in that column for easy movement.

Chapter 17: Assembler Directives

Introduction

An instruction is a line that tells the assembler how to build a machine language command for the microprocessor. An assembler directive tells the assembler itself to take some action. In some cases, this may involve reserving memory or setting up data tables for use by the program. Conditional assembly directives tell the assembler how to modify lines of source code, and what order to process them in. Other directives define the beginning and end of subroutines, assign values to labels, and perform various housekeeping functions.

Conditional assembly directives and macro language directives are covered in the next chapter.

Except for the operand field, an assembler directive is coded in the same way as an instruction. The operand field is used to tell the assembler directive what to do. Since there are a variety of assembler directives, there are a variety of types of operands.

Descriptions of Directives

In the descriptions below, each directive has a model line showing the format for the directive. Anything that appears in uppercase in the model must be typed exactly as shown. Entries shown in lowercase represent variables. These are described in the table below. If there are two or more choices, these are shown separated by a vertical bar (|). Optional entries are enclosed in square brackets. For example,

```
[lab]     ABSADDR ON|OFF              [comment]
```

indicates that the ABSADDR directive can take ON or OFF as an operand. An optional label can be coded. Finally, a comment can appear after the operand.

variable	description
comment	Zero or more characters intended to document the program. Comments are not assembled by

	the assembler, but they are processed by the macro processor, which expands any symbolic parameters in the comment field.
expression	An expression is a logical or mathematical expression that resolves to a number. Both labels and constants can be used in the same expression. For details, see page 128.
lab	A label. Labels start with an alphabetic character, underscore or tilde and are followed by zero or more alphanumeric characters, tildes and underscores. Only the first ten characters are significant.
opcode	One of the operation codes recognized by the assembler, e.g. LDA, START.
pathname	A full or partial path name. If ProDOS considers it legal, the assembler will accept it. For partial path names, the current prefix is used.
segname	The name of a segment, i.e. a name that has appeared as the label of a START or DATA directive.
special	An operand whose format cannot be described on a single line. See the text for details.
string	A sequence of ASCII characters. If the string contains a space, it must be enclosed in either single or double quote marks. Quote marks within quote marks must be doubled.

Note that directives whose operand is either ON or OFF have the default listed right after the name of the directive.

ABSADDR OFF — Show Absolute Addresses

```
[lab]     ABSADDR ON|OFF              [comment]
```

Since an assembler that produces object modules that are later linked by a linker can never be sure what absolute address code will be located at, ORCA/M does not usually try to show absolute addresses. When on, this directive gives the assembler's best guess at the final execution address. If the assembly is a full assembly, the code is not used as a library, and if an ORG is not later specified during the link step, these addresses will be correct. The addresses appear as a column of two digit hexadecimal numbers at the far left of the listing.

ALIGN — Align to a Boundary

```
[lab]     ALIGN  expression       [comment]
```

The ALIGN directive has two distinct uses, depending on where in the program the directive occurs. If it appears before a START or DATA directive, it tells the link editor to align the segment to a byte boundary divisible by the absolute number in the operand of the ALIGN directive. This number must be a power of 2. For example, to align a segment to a page boundary, use the sequence

```
          ALIGN 256
SEG       START
          END
```

Within a segment, ALIGN inserts enough zeros to force the next byte to fall at the indicated alignment. This is done at assembly time, so the zeros show up in the program listing. If align is used in a subroutine, it must also have been used before the segment, and the internal align must be to a boundary that is less than or equal to the external align.

APPEND — Append a Source File

```
          APPEND                    pathname [comment]
```

Processing is transferred to the beginning of the file pathname. Any lines following the APPEND directive in the original file are ignored.

COPY — Copy a Source File

```
        COPY  pathname          [comment]
```

Processing is transferred to the beginning of the file pathname. After the entire file is processed, assembly continues with the first line after the COPY directive in the original file. A copied file can copy another file; the depth is limited by the available memory, and is generally about three or four levels.

DATA — Define Data Segment

```
lab     DATA                    [comment]
```

The DATA directive is used instead of the START directive to define a special form of program segment which contains no instructions. Its purpose is to set up data tables which several subroutines can access. Its labels become local labels for any subroutine which issues a USING directive for the data segment. The name of the data segment is the label field of the DATA directive, and is global. No more than 127 data segments may be defined in any one program.

Labels within data areas should not be duplicated in other data areas.

DC — Declare Constant

```
[lab]   DC    special           [comment]
```

The DC directive is used for every type of program constant definition. The operand begins with an optional repeat count, which must be in the range 1 to 255 decimal. The variable being defined will be placed in the object file as many times as specified by the repeat count. Next comes an identifier describing the value type. This is followed by the value itself, enclosed in quote marks. The entire sequence can then be followed by a comma and another definition. For example,

```
LABEL   DC    2I'2,3',I1'4'
```

would place five integers into memory, four sixteen-bit and one eight-bit. The resulting hexadecimal values would be

```
02 00 03 00 02 00 03 00 04
```

There are several options available for defining data. These are listed separately below.

Integer

```
nI[x]'expression[,expression,...]'
nI1<'expression[,expression,...]'
nI1>'expression[,expression,...]'
nI1^'expression[,expression,...]'
```

Integers can be defined in a variety of lengths, where the length is specified by replacing x with a digit from 1 to 8. If omitted, two byte integers are generated. All integers are stored least significant byte first; this is the format used by the 65xx family of CPUs and the ORCA subroutine libraries. Integers of length one to four bytes can be expressed as expressions, including external references. Longer integers can only be expressed as a signed decimal number.

When one byte integers are selected, an additional format option is available. Right before the quote, the >, <, and ^ characters used to select bytes for immediate addressing can be used for the same purpose. See the examples below for a sample that illustrates this idea.

The table below gives the valid range of signed integers that can be expressed with each length of integer. The ORCA subroutine libraries contain subroutines to perform math operations on 2,4 and 8 byte integers.

```
Size                Smallest                Largest

1                      -128                   127
2                    -32768                 32767
3                  -8388608               8388607
4               -2147483648            2147483647
5             -549755813888          549755813887
6           -14073748355328        14073748355327
7        -36028797018963968     36028797018963907
8     -9223372036854775808   9223372036854775807

Code                     Value

DC   I'4'                04 00
DC   2I'3'               03 00 03 00
DC   I1'2,3'             02 03
DC   I'$ABCD'            CD AB
DC   I'100/3'            21 00
DC   I1>'$ABCD,$1234'    CD 34
DC   I5'3'               03 00 00 00 00
DC   2I3'1,2'            01 00 00 02 00 00
                         01 00 00 02 00 00
```

Address

```
nA[x]'expression[,expression,...]'
```

This is actually similar to integer, but is more mnemonic for the intended use of building tables of addresses. Address type DC statements are limited to generating one to four byte values.

```
Code

DC   A'AD1,AD2'
```

Reference an Address

```
R'label[,label,...]'
```

This generates a reference to an address in the object module without saving the address in the final program. This allows a program to note that a subroutine will be needed from the subroutine library without reserving storage for the subroutine address. In conjunction with S below, this allows

for the development of a p-system which loads and links only those parts of the p-system language needed by a particular program. This option is then used by the p-code instructions to insure that any library subroutines that will be needed to execute that instruction are linked.

Another use for this format is to force inclusion of library routines. Libraries are searched one time from front to back. Because of this search method, if a library routine calls another that occurred before it in the library, the second routine may not be automatically included. If this happens, DC R directives can be used to force inclusion of the appropriate subroutines.

Note that this directive does not take up space in the finished program.

```
Code

DC   R'LIBRARY'
```

Soft Reference

```
nS[x]'expression[,expression,...]'
```

This generates one to four bytes of storage for each address in the operand, but does not instruct the link editor to link the subroutines into the final program. If the subroutine is not linked, the binary program produced by the link editor will resolve to zero. This allows a table of addresses to be built, but only those subroutines requested elsewhere in the program (usually by an R type reference) have their addresses placed in the table. See the discussion of R, above.

```
Code                    Value

DC   S'MISSING,ADR1234' 00 00 34 12
```

Hexadecimal Constant

```
nH'hex-digit-or-blank[hex-digit-or-blank...]'
```

The string between the single quote marks may contain any sequence of hexadecimal digits (0-9 and A-F) and blanks. Embedded blanks are removed, and the hexadecimal value is stored unchanged. If there are an odd number of digits, the last byte is padded on the right with a zero:

```
Code                    Value

DC   H'01234ABCDEF'     01 23 4A BC DE F0
DC   H'1111 2222 3333'  11 11 22 22 33 33
```

Binary Constant

```
nB'0|1|blank[0|1|blank...]'
```

The string between the quote marks can contain any sequence of zeros, ones, and blanks. The blanks are removed, and the resulting bit values are stored. If a byte is left partially filled, it is padded on the right with zeros:

```
Code                    Value

DC   2B'01 01 01 10'    56 56
DC   B'111111111'       FF 80
```

Character String

```
nC'character-string'
```

The string enclosed in quote marks may contain any sequence of keyboard characters. If a quote mark is desired, enter it twice to distinguish it from the end of the string:

```
Code                    Value

DC   C'NOW IS THE TIME ...' 4E 4F 57 20 49 53 20 54
                        48 45 20 54 49 4D 45 20
                        2E 2E 2E
DC   C'NOW''S THE TIME' 4E 4F 57 27 53 20 54 48
                        45 20 54 49 4D 45
```

Normally, strings are stored with the high-order bit off, corresponding to the ASCII character set. If characters will be written directly to the Apple II screen, it will be desirable to have the high bit set. In that case, use the MSB directive to change the default.

Note that the double quote character (") can be used instead of single quotes. Use of the double quote is reserved for use in macros.

Floating Point

```
nF'float-number[,float-number...]'
```

Numbers are entered as signed floating point numbers, with an optional signed exponent starting with E. Embedded blanks are allowed anywhere except within a sequence of digits.

The number is stored as a four byte floating point number. Bit one is the sign bit, and is 1 for negative numbers. The next eight bits are the exponent, plus $7E. The exponent is a power of two. The remaining 31 bits are the mantissa, with the leading bit removed since it is always 1 in a normalized number. The mantissa is stored most significant byte to least significant byte. This format is compatible with the IEEE floating point standard, and is also used by the ORCA subroutine library. See the IEEE directive for an way to get five byte floating point numbers compatible with Applesoft.

Numbers can range from approximately 1E-38 to 1E+38. The mantissa is accurate to over seven decimal digits.

```
Code                    Value

DC   F'3,-3,.35E1,6.25 E-2' 40400000
                        C0400000
                        40600000
                        3D800000
```

Double Precision Floating Point

```
nD'float-number[,float-number...]'
```

This is identical to F, except that an eight-byte number is generated with an eleven bit exponent and a forty-eight-bit mantissa. Numbers can range from about 1E-308 to 1E+308. The mantissa is accurate to slightly more than 15 decimal digits. The exponent is stored most significant byte first.

```
Code                    Value

DC   D'3,-3,.35E1,6.25 E-2' 4008000000000000
                            C008000000000000
                            400C000000000000
                            3FF0000000000000
```

DS — Declare Storage

```
[lab]    DS    expression        [comment]
```

This directive is used to reserve sections of memory for program use. The operand is coded the same way as an absolute address for an instruction. The operand is resolved into a four byte unsigned integer, and that many bytes of memory are reserved.

EJECT — Eject the Page

```
[lab]    EJECT                   [comment]
```

When a printer is in use, this directive causes the output to skip to the top of the next page. This can be of help in structuring the output of long subroutines. The directive does not effect the code sent to the output file in any way.

END — End Program Segment

```
         END                     [comment]
```

The END directive is the last statement in a program segment or data area. It directs the assembler to print the local symbol table and delete the local labels from the symbol table.

ENTRY — Define Entry Point

```
lab       ENTRY                    [comment]
```

It may be desirable to enter a subroutine someplace other than the top of the subroutine. Use of the ENTRY directive allows a global label to be defined for that purpose. The label field of the ENTRY statement becomes a global label.

EQU — Equate

```
lab       EQU   expression         [comment]
```

The label is assigned the value of the operand instead of the location counter. This allows a numeric value to be assigned to a name, with the name to be used instead of the number in further operands.

Although the operand may contain labels, these labels must already have a value. If they do not, an error is generated. This is because the resulting value may be a zero page address. During the first pass, the assembler has no way of knowing this, since it could not resolve the equate. Instructions are assumed to be absolute addresses on the first pass, and two bytes are reserved. On the second pass, the equate would be resolved as zero page. The addresses would now occupy only one byte, and further addressing would be incorrect.

For the same reason, it is important that equates defining zero page addresses be defined before they are used.

ERR ON — Print Errors

```
[lab]     ERR   ON|OFF             [comment]
```

If LIST ON has been specified, errors are always printed, regardless of this flag. If LIST OFF has been specified, this flag allows error lines to still be printed. If turned off, errors are no longer printed, but the number of errors found will still be listed at the end of the assembly.

EXPAND OFF — Expand DC Statements

```
[lab]    EXPAND                    ON|OFF   [comment]
```

If turned on, this option causes all bytes generated by DC directives to be shown in the output listing, up to a maximum of sixteen bytes. Only four bytes of a DC directive can be displayed on a line, so the option defaults to OFF to save paper and patience. When the option is turned off, only the first four bytes of the generated code are shown with the output.

GEN OFF — Generate Macro Expansions

```
[lab]    GEN   ON|OFF             [comment]
```

If GEN is turned on, all lines generated by macro expansions are shown on the output listing. Each line generated by a macro has a + character to the left of the line. If GEN is turned off, only the macro call is printed in the assembly listing. Errors within the macro expansion are still printed, along with the line causing the error.

GEQU — Global Equate

```
lab      GEQU  expression        [comment]
```

This is identical to the EQU directive, except that the label is saved in the global symbol table. All program segments are then able to use the label. Labels defined via the GEQU directive are resolved at assembly time, not link edit time. They are included in the object module, so library routines can use global equates to make constants available to the main program.

IEEE ON — IEEE Format Numbers

```
[lab]    IEEE  ON|OFF             [comment]
```

In its default setting, DC directives with F and D operands generate numbers compatible with the IEEE floating point standard. If IEEE is turned off, F type DC directives will generate Applesoft compatible numbers. D type DC directives are not effected; they always generate IEEE double precision numbers.

INSTIME OFF — Show Instruction Times

```
[lab]    INSTIME ON|OFF              [comment]
```

When turned on, this directive causes the assembler to print the number of CPU cycles required to execute a line of code. The number appears as two characters to the left of the source line. The first character is the number of cycles normally required by the instruction. If the number of cycles can change due to some other cause, such as crossing a page boundary during indexing, an asterisk is shown after the number of cycles.

KEEP — Keep Object Module

```
          KEEP  pathname          [comment]
```

The assembled code is saved on disk as a relocatable object module, using the specified name as the root name. The link editor may then be used to generate an executable binary file. This directive may only be used one time, and must appear before any code generating statements.

LIST ON — List Output

```
[lab]    LIST  ON|OFF            [comment]
```

A listing of the assembler output is sent to the current output device. If the listing is turned off, the assembly process speeds up by about 10%.

LONGA ON — Accumulator Size Selection

```
[lab]    LONGA ON|OFF                [comment]
```

The 65816 CPU is capable of doing sixteen bit operations involving the accumulator and memory; it is also capable of performing eight bit operations the same way the 6502 and 65C02 do. The size of the accumulator and the amount of memory effected by instructions like LDA, STA and INC are controlled by a bit in the processor status register. At assembly time, the assembler has no idea how that bit will be set at run time

- it is the responsibility of the programmer to tell the assembler using this directive. LONGA ON indicates 16 bit operations, while LONGA OFF indicates eight bit operations. The only difference this will make in the assembled program is to change the number of bits placed in the code stream when an immediate load is performed. For example,

```
LONGA ON
LDA   #2                    2 byte operand
LONGA OFF
LDA   #2                    1 byte operand
```

The status bit that the CPU uses at run time must be set separately. To do so, use the REP and SEP instructions.

LONGI ON — Index Register Size Selection

```
[lab]    LONGI ON|OFF                [comment]
```

This directive controls the number of bytes reserved for immediate loads to the X and Y registers when using the 65816. See LONGA for a complete discussion.

MCOPY — Copy Macro Library

```
[lab]    MCOPY pathname          [comment]
```

The name of the file is placed in a list of available macro libraries. If an operation code cannot be identified, all macro files in the list are loaded into the macro buffer in sequence, and checked for a macro with the specified name. The search begins with the macro file in memory, proceeds to the first file in the list of macro files, and continues through to the last file in the list, in the order the respective MCOPY directives were encountered (skipping the one that was originally in memory). If no macro with a corresponding name is found, an error is generated.

No more than four macro libraries can be active at any one time. Macro libraries cannot contain COPY or APPEND directives.

MDROP — Drop a Macro Library

```
[lab]    MDROP pathname              [comment]
```

Removes pathname from the list of macro libraries. This might be necessary if more than four libraries are being used. It can also speed up processing if a library is no longer needed.

If the macro library is active at the time the MDROP directive is encountered, it is left there and searched for macros until a search is made which loads a different library, or until an MLOAD directive is used.

MEM — Reserve Memory

```
[lab]    MEM   expression,expression  [comment]
```

The operand for this directive is two absolute addresses, separated by a comma. The absolute addresses specify a range of memory that is to be reserved as a data area. The link editor will insure that subroutines are not placed in this range of memory. This is done by checking the length of each subroutine to see if it will enter a reserved area. If it does, it is started after the end of the reserved area. This directive is intended for use when the high resolution graphics pages are needed.

MERR — Maximum Error Level

```
[lab]    MERR  expression         [comment]
```

MERR sets the maximum error level that can be tolerated and still allow the assembled program to link edit immediately after the assembly (as would happen with a RUN command from the monitor). The default value is zero. The operand is evaluated to a one byte positive integer.

MLOAD — Load a Macro Library

```
[lab]    MLOAD pathname              [comment]
```

The list of macro libraries is checked. If pathname is not in the list, it is placed there. The file is then loaded into the macro library buffer.

This directive can be used to speed up assemblies by helping the macro processor to find macros.

MSB OFF Set the Most Significant Bit of Characters

```
[lab]    MSB   ON|OFF            [comment]
```

Character constants and characters generated by DC statements have bit seven cleared, corresponding to the ASCII character set. If MSB ON is coded, characters generated have bit seven turned on, and appear normal on the Apple screen display.

OBJ Set Assembly Address

```
[lab]    OBJ   expression        [comment]
```

Normally, the assembler assembles code as if it will be executed where it is placed in memory when loaded. When code will be moved before it is executed, the OBJ directive is used to tell the assembler where the code will be moved to. That way, the code can physically reside at one location, but be moved to another before execution. The most common reason for doing this is to install drivers that will remain in memory after the program finishes.

All code that appears after the OBJ directive is assembled as if it will be executed at the address specified in expression. This effect continues until another OBJ, an OBJEND, or an END is encountered.

OBJEND Cancel OBJ

```
[lab]    OBJEND                  [comment]
```

Cancels the effect of an OBJ. See OBJ for details.

ORG Origin

```
[lab]    ORG   expression        [comment]
[lab]    ORG   *+expression      [comment]
[lab]    ORG   *-expression      [comment]
```

When the assembly process starts, the assembler assumes that the program will be located at $2000. The ORG directive can be used to change that default and start the program at some other location. The location is specified as an absolute address in the operand field. To do this, place the ORG before the first START or DATA directive.

The ORG directive can also be positioned before any subsequent START or DATA directives to force that segment to a particular fixed address. Again, the operand is an absolute address, and must be a constant. In this case, though, the actual method of performing the ORG is to insert zeros until the desired location is realized. This action is performed by the link editor as the final binary module is built.

The ORG directive can also be used inside a program segment, but in that case the operand must be a *, indicating the current location counter, followed by + or -, and a constant expression. The location counter is moved forward or backward by the indicated amount. Thus,

```
ORG   *+2
```

is equivalent to

```
DS    2
```

while

```
ORG   *-1
```

deletes the last byte generated. It is not possible to delete more bytes than have been generated by the current segment.

PRINTER OFF — Send Output to Printer

```
[lab]     PRINTER                    ON|OFF    [comment]
```

If PRINTER ON is coded, output is sent to the printer. A printer capable of printing at least eighty columns is expected there. If a printer is not connected, the system will hang. The slot number and printer characteristics may be changed by re-configuring the operating system. If the option is turned off, output is sent to the video display.

RENAME — Rename Op Codes

```
[lab]     RENAME                     opcode,opcode
[comment]
```

ORCA is powerful enough to actually develop a cross assembler using macros. The only problem is that other CPU's may have an op code that conflicts with an assembly language instruction or an existing ORCA directive. This problem can be resolved by renaming the existing op code to prevent a conflict. The operand is the old op code followed by the new one. In the following example, the first time LDA is encountered, it is a 6502 instruction. The second time, it is not found in the op code table, so the assembler tries to expand it as a macro.

```
INST      START
          LDA   #1
          END

          RENAME                     LDA,NEW
MACRO     START
          LDA   #1
          END
```

Restrictions on the RENAME directive are that it cannot be used inside a segment (i.e., it cannot come between a START and END), the new op code name must be eight characters or less, and the op code name cannot contain spaces or the & character.

SETCOM — Set Comment Column

```
[lab]     SETCOM                    expression
[comment]
```

There is a column beyond which the assembler will not search for an op code, and will not search for an operand unless there is exactly one space between the op code and operand. This is customarily where comments are started, so that a comment is not accidentally used as part of an operand. This column defaults to forty, but can be changed to any number from one to eighty by specifying the number in the operand field of this directive.

START — Start Subroutine

```
lab       START                     [comment]
```

Each program segment (that is, both main programs and subroutines), must begin with a START directive. Labels defined inside a program segment are local labels, and are valid only inside the program segment that defined them. There is nothing wrong, for example, with having a local label called LOOP in every program segment in a source file.

Every START directive must have a label. This becomes a global label. Therefore, every program segment in the program is able to reference that subroutine, allowing it to be called or jumped to from any program segment (including itself).

The label on the START directive becomes the subroutine name in the object module that is the output of the assembler. Since it is a global label, the link editor can inform other subroutines of its location at link edit time. This allows subroutines that are assembled separately to be combined later by the link editor.

SYMBOL ON — Print Symbol Tables

```
[lab]     SYMBOL                    ON|OFF    [comment]
```

An alphabetized listing of all local symbols is printed following each END directive. After all processing is complete, global symbols are printed. If

this option is turned off, assemblies speed up slightly. The option can also be used to save paper.

TITLE — Print Header

```
[lab]    TITLE [string]              [comment]
```

The title directive has an optional operand. If coded, it must be a legal string, and must be enclosed in single quote marks if it contains blanks or starts with a single quote mark. If the string is longer than sixty characters, it is truncated to sixty characters.

If the TITLE directive is used, page numbers will be placed at the top of each page sent to the printer. If an operand was coded, the string used will be printed at the top of each page, immediately after the page number.

TRACE OFF — Trace Macros

```
[lab]    TRACE ON|OFF                [comment]
```

Most conditional assembly directives, covered in the next chapter, do not get printed by the assembler. This is to avoid line upon line of output that has no real effect on the finished program. Especially when debugging macros, it is desirable to see all of the lines the assembler processes. To do this, use TRACE ON.

USING — Using Data Area

```
[lab]    USING segname               [comment]
```

This statement should appear in any program segment that wants access to local labels within a given data area. The operand field contains the name of the data area. Labels defined within the subroutine take precedence over labels by the same name in data areas.

65C02 OFF — Enable 65C02 Code

```
[lab]     65C02 ON|OFF                [comment]
```

The 65C02 is used the Apple //c, and can be retrofitted to older Apples. The extra instructions and addressing modes available on that CPU can be enabled and disabled with this directive.

65816 OFF — Enable 65816 Code

```
[lab]     65816 ON|OFF                [comment]
```

When off, 65816 instructions and operands are identified as errors by the assembler, allowing 65C02 or 6502 code to be generated without fear of accidentally using a feature not available on the smaller CPU.

Chapter 18: Macro Language and Conditional Assembly Language

Introduction to Macros

This chapter describes how to create user-defined macros. It is not necessary to be able to write a macro in order to use one. It is therefore not necessary to know the material in this chapter in order to use the assembler; the macro language is an advanced capability, which should be studied after the fundamental features of the assembler have been mastered. In this chapter, all of the macro and conditional assembly language directives are covered in detail. Chapter 8 of the User's Manual also covers writing macros, but at an introductory level.

The Macro File

A new macro is created by coding a macro definition, which tells the assembler which instructions to replace the macro call with. These definitions are kept in a special file called a macro file. Macro files are created using the text editor in the same way that a source file is created. The distinction between the two is in the way the assembler handles them. Macro files are included in the source stream using MLOAD and MCOPY directives. The assembler loads them into a special area of memory called the macro buffer as they are needed. When an unidentified operation code is encountered in the source file, the assembler searches for a macro definition of that name in the macro buffer.

Writing Macro Definitions

There are three macro language directives: MACRO, MEND, and MEXIT. These directives are valid only in a macro file; if used inside of a regular source file, they will create an error.

Each macro definition begins with a MACRO directive and ends with an MEND directive. These directives are coded like an operation code. No operand or label is needed, and any present is ignored. Their sole purpose is to set the macro definition apart from others in the file. Their use will become clear in the examples that follow shortly.

Immediately following the MACRO directive is the macro definition statement. The name of the macro being defined is placed in the the operation code field. If the operation code that the assembler is trying to identify matches it, the assembler uses the definition that follows to replace the macro call in the source file with the instructions found in the body of the macro itself. The macro definition operation code may be any sequence of keyboard characters except blanks or the & character. It may contain any number characters, but only the first ten are significant.

Consider the following simple macro as an example. It is used to print a character on the screen. The Apple monitor has a subroutine called COUT which does this, but it is inconvenient to have to always remember its address. To remedy this, a macro may be defined, using the same name (COUT) by entering this definition in a new source file called MACROS:

```
            MACRO
            COUT
            JSR   $FDED
            MEND
```

COUT may now be used as a new instruction:

```
            MCOPY MACROS          COPY 'COUT' MACRO INTO
   !                              MACRO BUFFER
            LDA   #'A'
            COUT                  PRINT CHARACTER IN
   !                              ACCUMULATOR ("A")
```

Following the COUT instruction, the assembler includes the macro expansion, yielding the sequence

```
             .
            .
            .
            LDA   #'A'
            COUT
      +           JSR   $FDED
            .
            .
            .
```

in the output listing. The new instruction COUT may be used as many times as desired anywhere in a source program provided the MCOPY

MACROS directive is also included. The assembler always prints the name of the macro (in this case COUT) to show how instructions that follow were generated; the JSR instruction is the only part of the macro expansion that actually generates code. The + character at the beginning of the line containing the JSR instruction is put there by the assembler to indicate that the line was generated by the macro processor, rather than coded directly by the programmer. The lines that comprise the macro expansion are normally not printed in the assembly listing; a GEN ON directive must be issued earlier in the program to list the macro expansion.

The statements between the macro definition statement and the MEND directive are called model statements, since the macro processor uses them as models for the new instructions. The instruction in the source file that caused the macro to be expanded is called the macro call statement, or simply the macro call.

Macros may contain references to other macros, up to four levels deep. They cannot contain COPY or APPEND directives.

Symbolic Parameters

A symbolic parameter is a special variable used by the assembler. Unlike labels, they are true variables; that is, they may be assigned a value which can later be changed. They come in three kinds: A for arithmetic, B for boolean (logical) and C for character type.

A symbolic parameter is coded as an & character followed by the symbolic parameter name. The name itself has the same syntax conventions as a label.

When the assembler encounters a symbolic parameter, it replaces it with its value before assembling the line. The value may be set in several ways. One way, described below, is by passing the values during the macro call. Only character type symbolic parameters may be passed this way; the use of the other types of symbolic parameters will be explained later, in the section covering conditional assembly.

Positional Parameters

One way to define a character type symbolic parameter is to include it in the label or operand field of a macro definition statement. Symbolic parameters defined in this way may be said to be implicitly defined by appearing on the

macro definition line. Character type symbolic parameters defined in this way are used to pass actual values to the symbolic parameters during a macro call, as will be seen in the example below. Revisiting the macro defined above to output a character, a new, more powerful macro definition may be written which reads

```
          MACRO
&LAB      COUT  &CHAR
&LAB      LDA   &CHAR
          JSR   $FDED
          MEND
```

This new macro is called from a source program as follows.

```
          .
          .
          .
          BEQ   L1
          COUT  #'A'
          JMP   L2
L1        COUT  #'B'
L         RTS
          .
          .
          .
```

At assembly time, the following code is generated. Note again that the assembler includes the macro call statement only to show what generated the new lines; there is no generated code associated with the macro call line itself:

```
            .
            .
            .
            BEQ   L1
            COUT  #'A'
+           LDA   #'A'
+           JSR   $FDED
            JMP   L2
  L1        COUT  #'B'
+ L1        LDA   #'B'
+           JSR   $FDED
  L2        RTS
            .
            .
            .
```

The reason that &CHAR is referred to as a positional parameter is that it gets its value by being matched with a character string in the source file by position. This becomes clear when a macro is defined which has two or more symbolic parameters. Also note that the symbolic parameter defined in the label field of the macro definition (&LAB) resulted in the label field of the first line of the macro expansion receiving the value of L1 after the second macro call. The symbolic parameter &LAB was also coded in the first line of the macro body, where the value of the macro call label field was substituted for it during the macro expansion. Note that if &LAB had been omitted from either place in the macro, L1 would not have been defined and the BEQ L1 statement would have generated an error.

The following example, which is a macro to print two characters, shows how positional parameters are set via the macro call:

```
            MACRO
  &LAB      COUT2 &C1,&C2
            LDA   &C1
            JSR   $FDED
            LDA   &C2
            JSR   $FDED
            MEND
```

Observe that the two symbolic parameter declarations on the macro definition line were separated by a comma, with no intervening spaces. The comma delimits the different positional parameters; spaces are not allowed. When the macro is called, as shown below, the actual parameters are coded

identically, that is, with commas separating the fields, and no intervening blanks:

```
            .
            .
            .
            COUT2 #'A',#'B'
+           LDA   #'A'
+           JSR   $FDED
+           LDA   #'B'
+           JSR   $FDED
            .
            .
            .
```

The macro processor determined which actual parameters to substitute for which symbolic parameters by matching their relative positions in the macro call statements with those in the macro definition.

Once conditional assembly instructions are introduced below, it will be seen that there are times when a positional parameter may (optionally) not be coded. In this case, nothing need be coded in the source file. However, all commas must be included, as if something had been coded. The macro processor keeps count of the position using the commas, so that later positional parameters appear in the right place.

Keyword Parameters

A keyword parameter is another way to reference a symbolic parameter defined in the operand field of a macro statement. The name of the symbolic parameter is typed, followed by an equal sign, and the value to assign it. For example, a call to the COUT2 macro could be coded as

```
        .
        .
        .
        COUT2 C2=#'B',C1=#'A'
+       LDA   #'A'
+       JSR   $FDED
+       LDA   #'B'
+       JSR   $FDED
        .
        .
        .
```

When keyword parameter substitution only is used, the order is not important. The same rules as for positional parameters regarding commas and blanks do apply, however. Keyword and positional parameters can be mixed. If this is done, keyword parameters take up a space, and are counted for determining positions. The macro processor simply counts the number of commas encountered when setting values for positional parameters. Note that the & character is not coded when referencing a parameter as a keyword.

Subscripting Parameters in Macro Call Statements

All types of symbolic parameters may be subscripted. Character type symbolic parameters defined in the macro definition statement are subscripted by including the subscripted variables in parentheses on the macro call line. For example, if a macro call statement contained the following phrase in the operand field

```
SUB=(ALPHA,,GAMMA)
```

(an example of keyword parameter substitution), the symbolic parameter &SUB for the given expansion would have three subscripts allowed. The initial value of each element would be:

```
&SUB(1)                    'ALPHA'
&SUB(2)                    null string
&SUB(3)                    'GAMMA'
```

To effectively use subscripted actual parameters, the macro itself would have to be coded so as to detect the number of subscripts allowed and to take appropriate action via conditional assembly directives.

Explicitly Defined Symbolic Parameters

The use of character type symbolic parameters in the macro definition and macro call lines was explained above.

In addition to being defined implicitly in the macro definition statement, as in the case of character-type symbolic parameters, all symbolic parameter types (arithmetic, boolean, and character) may be declared explicitly.

Explicitly defined symbolic parameters are not set with actual parameters via a macro call. Rather, they are used as internal variables within a macro or source file. Using the conditional directives, their values may be set and reset during the macro expansion, resulting in an extremely powerful conditional assembly capability.

Explicitly defined symbolic parameters may also be subscripted. The subscript must follow the symbolic parameter name. Only a single subscript is allowed. The range that is acceptable depends on the type of the symbolic parameter. See the descriptions of LCLA, LCLB, LCLC, GBLA, GBLB and GBLC below for details. A symbolic parameter used as a subscript for another symbolic parameter cannot be subscripted.

In the following example, assume that four symbolic parameters have been defined, as listed below. The maximum allowable subscripts for the subscripted symbolic parameters are shown with the symbolic parameter name. Next is the type, followed by the value. Subscripted symbolic parameters have their values listed on successive lines.

```
name           type      value(s)

&ART           A         $FE
&BIN(2)        B         1  (true)
                         0  (false)
&CHAR          C         'LABEL'
&CHAR2(3)      C         'STRING1'
                         ''  (null string)
                         'A'
```

Below are instructions as typed in a macro file, on the left, with the instructions as expanded by the macro processor on the right.

```
&CHAR     LDA   &CHAR2(1)     LABEL      LDA   STRING1
          STA   &CHAR.&BIN(2)            STA   LABEL0
          LDA   #&ART                    LDA   #254
          BEQ   L&BIN                    BEQ   L1
          LDA   LB&CHAR2(2)              LDA   LB
L&BIN(1)  STA   EQ&BIN(2)     L1         STA   EQ0
          LD&CHAR2(3) #1                 LDA   #1
```

Note that a boolean symbolic parameter becomes zero if false and one if true. The null string is valid; it is replaced by nothing.

The second line demonstrates the use of the period to concatenate symbolic parameters. The period itself does not appear in the final line. It can be used after any symbolic parameter, regardless of how that parameter was defined. It must be used if a symbolic parameter is followed by a character, or if a subscript is followed by a mathematical symbol or expression.

Symbolic Parameter Definition Statements

Symbolic parameters may be defined either for the current macro expansion or for the entire subroutine. Defining symbolic parameters whose scope is the entire subroutine allows macros to communicate with each other. Symbolic parameters which are only valid inside a macro are called local symbolic parameters; those valid throughout the subroutine are called global symbolic parameters.

A symbolic parameter definition statement does not contain a label. The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, the maximum allowable subscript must be specified in parentheses immediately following

Symbolic parameter definition statements are not printed in the output listing unless they contain errors or TRACE ON has been coded.

A permanent global symbolic parameter called &SYSCNT of type arithmetic is available. Its value is set to one at the beginning of each subroutine and is incremented at the beginning of each macro expansion. It is used to prevent labels defined inside macros from being duplicated if the same macro is used more than once in the same subroutine. This is done by concatenating &SYSCNT to any labels used within the macro definition itself.

```
Examples:

LCLA  &NUM
GBLC  &STRINGS(40)
```

Sequence Symbols

The conditional assembly branch instructions AGO and AIF must have someplace to go. This is provided by sequence symbols.

A sequence symbol is a line with a period in column one, followed by a label. Comments may follow the label after at least one space. Instructions contained in the line are treated as comments. The line is not printed in the output listing unless TRACE ON is used.

Attributes

In certain cases it is desirable to know something about a label or symbolic parameter other than its value. This information is provided via attributes, which may be thought of as functions that return information about a label or symbolic parameter.

An attribute is coded as an attribute letter, a colon, and the label or symbolic parameter it is to evaluate. For example, the length attribute of the label LABEL is coded as

```
L:LABEL
```

Attributes may be used in operands in the same way that a constant is used.

C: Count

The count attribute gives the number of subscripts defined for a symbolic parameter. It is normally used to find out if a multiple argument has been assigned to a symbolic parameter by a macro call. It can also be used to find out if a symbolic parameter (or label) was defined at all; if not, the count attribute is zero. The count attribute of a defined label is one.

L: Length

The length attribute of a label is the number of bytes generated by the line that defined the label.

The length attribute of an arithmetic symbolic parameter is four. For a boolean symbolic parameter it is one. For a character symbolic parameter, it is the number of characters in the current string. If the symbolic parameter is subscripted, the subscript of the desired element should be specified; otherwise, the first element is assumed.

S: Setting

Setting is a special attribute that returns the current setting of one of the flags set using directives whose operand is ON or OFF. If the current setting is ON, the result is 1, otherwise the result is 0. For example, if it were necessary to write a macro which expanded to two different code sequences depending on whether the accumulator was set to 8 or 16 bits on a 65816, one could use S:LONGA to test the current setting of the LONGA directive. The directives which accept ON or OFF for operands are summarized below.

```
LIST        LONGA       PRINTER
SYMBOL      LONGI       MSB
ERR         65816       IEEE
GEN         65C02       TRACE
EXPAND
```

T: Type

The type attribute evaluates as a character. The type attribute of a label indicates the type of the operation in the line that defined the label. For a symbolic parameter, the type attribute is used to distinguish between A, B and C type symbolic parameters. The character that is returned for each type is indicated in the table below.

Character	Meaning
A	Address Type DC Statement
B	Boolean Type DC Statement
C	Character Type DC Statement
D	Double Precision Floating Point Type DC Statement
F	Floating Point Type DC Statement
G	EQU or GEQU Directive
H	Hexadecimal Type DC Statement
I	Integer Type DC Statement
K	Reference Address Type DC Statement
L	Soft Reference Type DC Statement
M	Instruction
N	Assembler Directive
O	ORG Statement
P	ALIGN Statement
S	DS Statement
X	Arithmetic Symbolic Parameter
Y	Boolean Symbolic Parameter
Z	Character Symbolic Parameter

If a DC statement contains more than one type of variable, the first type in the line determines the type attribute.

It is worth noting that many assembler directives are not printed, yet they do have a label field. Generally, it is not a good idea to put labels in this field, since the line will not be found in the output listing.

Examples of macro definitions and how they use conditional assembly directives can be found in the macro library.

Conditional Assembly and Macro Directives

ACTR — Assembly Counter

```
[lab]    ACTR expression            [comment]
```

Each time a branch is made in a macro definition, a counter is decremented. If it reaches zero, processing of the macro stops, to protect from infinite loops.

The ACTR directive is coded with a number from 1 to 255 in the operand field. The counter is then assigned this value. The ACTR directive is used to limit the number of loops caused by conditional assembly branches. In loops with more than 255 iterations, it must be reset within the body of the loop to prevent the counter from reaching zero.

The counter value is set to 255 automatically at the beginning of each macro.

The ACTR directive is not printed unless it contains an error or TRACE ON is used.

AGO Unconditional Branch

```
[lab]     AGO seq-symbol              [comment]
```

The operand contains a sequence symbol. The macro definition (or subroutine, if not used in a macro) is searched for a matching sequence symbol. Processing continues with the instruction immediately following the sequence symbol.

The search range for a source file includes the entire file, not just the subroutine containing the AGO directive. Searching begins in the forward direction and continues until the sequence symbol is found or the end of the file is reached. The search then begins with the instruction before the AGO directive and continues toward the beginning of the file.

The search process in a macro definition is similar, except that the search will not cross an MEND or MACRO directive.

Searches for sequence symbols will not cross into a copied or appended file; they are limited to the file in memory.

The AGO directive is not printed in the output listing unless it contains an error or TRACE ON is used.

In the following example, the assembler encounters the initial AGO directive. Processing continues at the sequence symbol. All lines between the AGO and sequence symbol are ignored by the assembler.

```
            AGO   .THERE
!   THESE LINES ARE IGNORED.
            .
            .
            .
.THERE
```

Processing branches is one of the most time consuming tasks performed by the assembler. For that reason, it should be kept in mind that when looking for a sequence symbol, the assembler searches forward first, then backward. If a sequence symbol appears before the branch, code a ^ character instead of a period for the first character of the sequence symbol. This forces the assembler to skip the forward search, proceeding directly to the backward search. For example,

```
            AGO   ^THERE
```

will not search forward at all, but will search backward in the file.

AIF Conditional Branch

```
[lab]    AIF expression,seq-symbol          [comment]
```

The operand contains a boolean expression followed by a comma and a sequence symbol. The boolean expression is evaluated. If true, processing continues with the first statement following the sequence symbol; if false, processing continues with the first statement following the AIF directive. As with the AGO directive, the . in the sequence symbol may be replaced with a ^ character to speed up branches in the case where the destination sequence symbol comes before the AIF directive.

The AIF directive is not printed in the output listing unless it contains an error or TRACE ON is used.

As an example, consider a file which contains the following statements

```
          LCLA  &LOOP
&LOOP     SETA  4
.TOP
          ASL   A
&LOOP     SETA  &LOOP-1
          AIF   &LOOP>0,.TOP
```

The output listing will contain these lines:

```
          ASL   A
          ASL   A
          ASL   A
          ASL   A
```

While the above example is very straight forward, there is a more efficient way to code it. Coding for efficiency, the loop would be

```
          LCLA  &LOOP
&LOOP     SETA  4
.TOP
          ASL   A
&LOOP     SETA  &LOOP-1
          AIF   &LOOP,^TOP
```

The first difference, which appears in the last line, is that we have used the ^ character on the symbolic parameter to indicate that the label occurs before the AIF statement. This allows the assembler to skip searching for .TOP in the forward direction, saving a great deal of time. A smaller savings is also realized in the same statement by depending on the fact that any non-zero value is treated as true in a logical expression. The branch will be made as long as &LOOP is non-zero.

AINPUT — Assembler Input

```
sym-parm AINPUT [string]                    [comment]
```

The operand is optional and, if coded, consists of a literal string. If the operand is coded, the string contained in the operand is printed on the screen during pass one as an input prompt. The assembler then waits for a line to be entered from the standard input (usually the keyboard, but it can be redirected from a text file). The string entered is assigned to the character type symbolic parameter specified in the label field.

During pass one, keyboard responses are saved by the assembler. When an AINPUT directive is encountered on pass two, the response given in pass one is again placed in the symbolic parameter specified in the label field. Thus, keyboard response is only needed one time for each input, but the symbolic parameter is set to the response on both pass one and pass two. This means that it is safe to use the response for conditional branching.

```
Example:

&RESULT AINPUT  'Prompt: '
```

AMID Assembler Mid String

```
sym-parm AMID string,expression,expression
                                                [comment]
```

This is a special kind of character type set symbol which provides a mid-string function. It has three arguments in the operand field, separated by commas. Embedded blanks are not allowed.

The first argument is the string to be operated on. It must be a simple string (no concatenation is allowed). If the string contains embedded blanks or commas, it must be enclosed in quote marks. Quote marks inside quote marks must be doubled.

The second and third arguments are of type arithmetic. The second argument specifies the position within the target string of the first character to be chosen. It must be greater than zero. Characters from the target string are numbered sequentially, starting with one. The third argument specifies the number of characters to be chosen.

If the combination of the last two arguments result in an attempt to select characters after the last character of the target string, the selection is terminated. Characters already selected are still valid.

The resulting string is assigned to the character type symbolic parameter specified in the label field.

```
Examples:

instruction                          resulting string

&CHAR    AMID  'TARGET',2,3          ARG
&CHAR    AMID  'TARGET',5,3          ET
&CHAR    AMID  'TARGET',7,3          null string
```

ANOP Assembler No Operation

```
[lab]    ANOP                    [comment]
```

The ANOP directive does nothing. It is used to define labels without an instruction. The label assumes the current value of the program counter.

ASEARCH Assembler String Search

```
sym-parm ASEARCH string,string,expression
                                 [comment]
```

This is a special form of arithmetic set symbol. It implements a string search function for character type symbolic parameters.

The ASEARCH directive has three arguments. The first is of type character, and is the target string to be searched. The second is also of type character, and is the string to search for. The last is of type arithmetic, and is the position in the target string to begin the search. The search can be conducted for any sequence of keyboard characters.

The label field contains an arithmetic symbolic parameter. It is set to the character position in the target string where the search string was first found. If the search string was not found, it receives the value zero.

```
Examples:

instruction                        resulting value

&NUM     ASEARCH 'TARGET',GE,1    4
&NUM     ASEARCH 'TARGET',GE,5    0
&NUM     ASEARCH 'TARGET',X,1     0
```

GBLA Declare Global Arithmetic Symbolic Parameter

```
          GBLA sym-parm              [comment]
```

Defines the arithmetic symbolic parameter sym-parm. The symbolic parameter is valid for the rest of the segment that it is defined in, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range one to sixty.

```
Examples:

          GBLA  &NEW
          GBLA  &NUM(45)
```

GBLB Declare Global Boolean Symbolic Parameter

```
          GBLB sym-parm              [comment]
```

Defines the boolean symbolic parameter sym-parm. The symbolic parameter is valid for the rest of the segment that it is defined in, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range 1 to 240.

Examples:

```
GBLB  &BOOL
GBLB  &ARR(45)
```

GBLC Declare Global Character Symbolic Parameter

```
GBLC sym-parm              [comment]
```

Defines the character symbolic parameter sym-parm. The symbolic parameter is valid for the rest of the segment that it is defined in, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range 1 to 240.

Examples:

```
GBLC  &CHAR
GBLC  &STRARR(45)
```

LCLA Declare Local Arithmetic Symbolic Parameter

```
LCLA sym-parm              [comment]
```

Defines the arithmetic symbolic parameter sym-parm. The symbolic parameter is valid only in the macro in which it is defined, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range one to sixty.

Examples:

```
LCLA  &NUM
LCLA  &ARR(45)
```

LCLB Declare Local Boolean Symbolic Parameter

```
LCLB sym-parm                [comment]
```

Defines the boolean symbolic parameter sym-parm. The symbolic parameter is valid only in the macro in which it is defined, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range 1 to 240.

Examples:

```
LCLB  &BOOL
LCLB  &ARR(45)
```

LCLC Declare Local Character Symbolic Parameter

```
LCLC sym-parm                [comment]
```

Defines the character symbolic parameter sym-parm. The symbolic parameter is valid only in the macro in which it is defined, including inside macros and in the source file itself.

The symbolic parameter can be declared as subscripted by following the name with a number inclosed in parentheses. If used, the subscript must be in the range 1 to 240.

Examples:

```
LCLC  &STR
LCLC  &ARR(45)
```

MACRO Start Macro Definition

```
        MACRO                   [comment]
```

The MACRO directive marks the start of a macro definition. It can be used only in a macro file. See the discussion at the beginning of the chapter for details on its use.

MEND End Macro Definition

```
        MEND                    [comment]
```

The MEND directive marks the end of a macro definition. It can be used only in a macro file. See the discussion at the beginning of the chapter for details on its use.

MEXIT Exit Macro

```
        MEXIT                   [comment]
```

An MEXIT directive indicates that a macro expansion is complete. Unlike MEND, it does not indicate the end of the macro definition. A good way conceptualize this directive is to think of it as a return from a macro definition. The MEND is the end of the definition, but the MEXIT can return from within the macro definition.

MNOTE Macro Note

```
[lab]   MNOTE string[,expression]       [comment]
```

A macro definition may include an MNOTE directive. The operand of an MNOTE directive contains a message, optionally followed by a comma and a number. The assembler prints the message on the output device as a separate line. If the number is present, it is used as a severity code for an error.

Assume that the following statements appear in a program:

```
*  MNOTE FOLLOWS
          MNOTE 'ERROR!',4
```

The output would look like this:

```
0432 10FE*  MNOTE FOLLOWS
ERROR!
```

Assuming that there were no other errors in the assembly, the maximum error level found (printed at the end of the assembly) would be four.

MNOTE is designed for use when conditional assembly directives are used to scan parameters passed via a macro call for correct (user defined) syntax. Although MNOTE statements are intended for use inside macros, they are legal inside of a source program.

SETA Set Arithmetic

```
sym-parm SETA expression                    [comment]
```

The operand field is resolved as a four byte signed hexadecimal number and assigned to the symbolic parameter in the label field.

```
Examples:

&NUM      SETA  4
&N(&NUM) SETA  &NUM2+LABEL*4
```

SETB Set Boolean

```
sym-parm SETB expression                    [comment]
```

The operand field contains a boolean expression, which is evaluated as true or false. If true, the symbolic parameter is assigned a value of one. If false, or if the line contains an error, the symbolic parameter is assigned a value of zero.

The boolean expression in the operand field for a SETB directive is coded using the same rules as an absolute address. It is referred to as a boolean

phrase because it most generally takes on a value of true or false (one or zero).

Recall that boolean operators may be used in expressions. If they are used, the resulting expression has a boolean value that appears as a zero or one used to indicate false and true boolean results. Arithmetic results are also valid in a boolean expression; thus a boolean variable can be used in the same way as arithmetic variables. Since only one byte is reserved for each boolean value, the boolean variable selects the least significant byte of an arithmetic result, using it as an unsigned arithmetic value in the range 0 to 255. Use of such a result in a boolean statement will result in the value being evaluated as true if the value is non-zero, and false if the value is zero.

```
Examples:

&FLAG    SETB  A<&NUM
```

SETC Set Character

```
sym-parm SETC string                          [comment]
```

The operand is evaluated as a character string and assigned to the symbolic parameter. Several sub-strings may be concatenated to make up the final string; they are separated in the operand field by plus characters (+). Such strings must be enclosed in quote marks. Embedded blanks outside of strings are not allowed. Quote marks inside quote marks must be doubled.

```
Examples:

&STRING(4) SETC &STRING
   &STR   SETC   '&FKENAME'+'.OBJ'
```

Chapter 19: Macro Libraries

The ORCA 4.1 macro and subroutine libraries provide a comprehensive set of primitive commands that greatly extend the instructions available to the assembly language programmer. Because of this extensive library, most programmers will never need to write a macro; instead, appropriate macros are selected from the macro libraries for use in a program. Macros that require utility subroutines will generate external references which will be automatically resolved by the link editor from the subroutine library.

Since the system is fairly automatic, the macros can be learned as if they were simply extensions to the 6502 instruction set. The typical steps involved in using the libraries would be:

1. Write a program conforming to the rules outlined in the macro descriptions.

2. Run the program through the MACGEN utility to create a unique, tailored macro library for the program.

3. Add an MCOPY directive at the beginning of the program for the file created by MACGEN.

4. Assemble and execute the program in the normal way.

The documentation of the macros is divided into six sections. The rest of this one discusses topics of general interest to all (or most) of the macros. This includes definition of the data formats used by the macros, as well as the four addressing modes which are common throughout the macro libraries. The last five sections discuss the macros themselves. Macros are discussed by topic. The first section deals with the mathematics macros, the second with input and output, the third with ProDOS support, the fourth with graphics, and the last section with those macros that did not fit into one of the previous sections.

Addressing Modes

Like the instruction set of the CPU, macros use a variety of addressing modes to increase the power and flexibility of each macro. There are four addressing modes supported by the macros; immediate, absolute, indirect and stack.

Immediate addressing is available on most macros that require an input to perform their function. An immediate operand is coded as a pound sign (#) followed by the value for the operand. All data types are supported. For example,

```
PUT8  #4500000000
```

would write the approximate population of the Earth to the screen.

Absolute addresses are coded as a number, label, or expression, using the same rules as absolute addresses on instructions. An absolute address designates the memory location to use as a source or destination by the macro.

Indirect addresses take the form of an address which points to the address of the data rather than the data itself. Indirect addressing is indicated by enclosing the absolute address where the effective address is stored in soft brackets. Thus,

```
MUL4  {P1},{P2}
```

multiplies the number pointed to by P1 by the one pointed to by P2, placing the result where P1 points.

Stack addressing refers to taking a source value from the "evaluation stack", or storing a result there. The evaluation stack is the stack used by the ORCA high level languages to pass parameters and evaluate expressions. It is a software stack, distinct from the hardware stack in page 1 for the 6502. The INITSTACK macro can be used to set up this stack. Further use is made using any of the I/O, math or move macros. A stack operand is indicated with an asterisk (*).

When discussing stack operations, it is customary to refer to values based on the "top of stack" (TOS). Thus, the value on the top of the stack is said to be at TOS, while the number "below" the top one is at TOS-1. With this in mind, the following operation divides the two byte integer at TOS-1 by the one at TOS, placing the result at TOS:

```
DIV2  *,*
```

This addressing mode is very convenient for reverse polish notation calculation, and must be used when doing parameter passing to high level languages. For any other application, it will probably not be useful.

Data Types

The macros support several data types, including three lengths of integers, characters, strings, and boolean variables. Typing is not enforced; it is possible to read a four byte integer into an area, then access it as a two byte integer. The type of data in use is indicated by a single character from the table below. This character is used as a part of the macro name. For example, the PUTx macro can be used to write any of these variable types to an output device; the type is indicated by replacing the x with one of the characters. Thus, PUT2 writes a two byte integers, while PUTS writes a string.

Character	Type
2	two byte integer
4	four byte integer
8	eight byte integer
C	character
S	string
B	boolean

The general convention of using a lowercase character in the macro name to represent a group of very similar macros is followed throughout the descriptions of the macros. This saves a great deal of space, makes the task of learning the macros easier, and serves to connect macros that might otherwise be scattered across the manual. Lowercase letters are never used in the name of a macro for anything else. Of course, your own macros can indeed use lowercase letters in the macro name.

Two Byte Integers

As the name implies, two byte integers require two bytes of storage each. Two's complement notation is used, with the least significant byte stored first, followed by the most significant. Two byte integers range from -32768 to 32767.

Four Byte Integers

Four byte integers require four bytes of storage. They are represented in two's compliments notation with the least significant byte stored first, proceeding sequentially to the most significant byte, which is stored last. The range represented by four byte integers is -2147483548 to 2147483647.

Eight Byte Integers

Eight byte integers require eight bytes of storage, are stored in two's complement notation, and are represented with the least significant byte first, proceeding to the most significant byte. The range of the eight byte integers is -9223372036854775808 to 9223372036854775807.

Character

A character requires one byte for storage. The ASCII character set is used to represent characters; in general, it doesn't matter if the high bit is on or off. The system provides all inputs with the high bit off, and converts any outputs as needed; the only conflict arises for comparisons. For that reason, it is recommended that character data always be represented with the high bit off.

Control characters have differing effects on various output devices. If an output device cannot respond to a given control character because that character is not defined, the control character is ignored. Check the technical descriptions of individual hardware devices for details. All terminal control codes are respected by the ORCA CRT drivers.

Strings

Strings are variable length sequences of characters. Each string is made up of three parts. The first part, which requires one byte, contains the maximum number of characters in the string; this can range from 1 to 254. The next byte contains the number of characters currently in the string; this ranges from 0 to the value of the first byte. The third field contains the characters in the string itself. One byte is reserved in this field for each possible character in the string; unused bytes are not defined and have unreliable values.

Strings require two bytes more than the maximum number of characters in the string for storage.

Boolean Variables

Boolean variables require one byte of storage. They are either TRUE (non-zero) or FALSE (zero).

Memory Usage

The macros may use one or more of the locations $80 to $FC of zero page. The ORCA operating system uses $FD to $FF. So long as the evaluation stack is not being used, these macros do not depend on the values of these zero page locations remaining unchanged: a program can also use these locations, so long as it, too, does not mind if the locations are disturbed.

The input macros do not use the GETLN subroutine of the F8 ROM, so they do not modify page 2. Naturally, the output macros do modify pages 4 to 7, since that is where the Apple display screen is located.

Page 1 is the hardware stack, and is subject to use by the macros.

All other memory locations used are documented in the Apple ProDOS reference manual, and are only used by the ProDOS macros.

Chapter 20: Mathematics Macros

The mathematics macros provide support for the three formats of integer numbers described earlier. When dealing with numbers, it will also be useful to look through the miscellaneous macros, which have number conversion macros and several macros which deal with two byte integers as unsigned numbers.

The macros in this section are contained in the following libraries:

```
M6502.INT2MATH
M6502.INT2MATH2
M6502.INT4MATH
M6502.INT8MATH
```

The macros described are:

```
ABSx  N1,N2          absolute value function
ADDx  N1,N2,N3       add
CMPx  N1,N2          signed compare
DIVx  N1,N2,N3       divide
MODx  N1,N2,N3       remainder function
MULx  N1,N2,N3       multiply
RANx  N1             random number
SIGNx N1,N2          sign function
SQRTx N1,N2          square root function
SUBx  N1,N2,N3       subtract
```

ABSx Integer Absolute Value

Forms:

```
LAB     ABS2    NUM1,NUM2
LAB     ABS4    NUM1,NUM2
LAB     ABS8    NUM1,NUM2
```

Operands:

LAB - Label.
NUM1 - The argument.
NUM2 - The result.

Description:

The result is the absolute value of the argument. NUM2 is optional; if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of all registers are lost. No errors are possible.

Coding Examples:

```
     ABS4  NUM1,*         places the absolute value of
!                          NUM1 on the software stack
     ABS2  {P1}           replaces the number pointed to
!                          by P1 with its absolute value
     ABS8  #10000000,NUM1 places the value of 10,000,000
!                          in NUM1
```

ADDx Integer Addition

Forms:

```
LAB     ADD2   NUM1,NUM2,NUM3
LAB     ADD4   NUM1,NUM2,NUM3
LAB     ADD8   NUM1,NUM2,NUM3
```

Operands:

LAB - Label.
NUM1 - The first argument for the addition.
NUM2 - The second argument for the addition.
NUM3 - The result of the addition.

Description:

A signed integer addition is performed on the two arguments, NUM1 and NUM2. NUM3 is optional; if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of the accumulator are lost. If any operand used indirect or stack addressing, the contents of the Y register are also lost. If the operation is on four or eight byte integers, the X and Y registers are both lost. For a two byte add, if an overflow occurs, the V flag is set, otherwise it is cleared. For four and eight byte operations, an overflows will set the overflow error condition. The description of the ERROR macro tells how to detect and report this error.

Coding Examples:

```
     ADD2  *,*           performs a stack addition,
!                         placing the result on the stack
     ADD4  {P1},NUM1     adds the four byte integer
!                          pointed to by P1 to the four
!                          byte integer at NUM2, saving
!                          the result in the location
!                          pointed to by P1
     ADD8  NUM1,#4       adds 4 to the eight byte integer
!                           at NUM1
     ADD2  NUM1,NUM2,NUM3  adds the two byte integers at
!                           NUM1 and NUM2, saving the
!                           result at NUM3
```

CMPx Integer Compare

Forms:

```
LAB     CMP2    NUM1,NUM2
LAB     CMP4    NUM1,NUM2
LAB     CMP8    NUM1,NUM2
```

Operands:

LAB - Label.
NUM1 - The first argument.
NUM2 - The second argument.

Description:

The first signed integer is compared to the second signed integer. The C and Z flags are set in the same way that they are set for CMP instructions; C is set if NUM1 >= NUM2 and cleared otherwise, and Z is set if NUM1 = NUM2 and cleared otherwise. Branch instructions and branch and conditional jump macros can be used after the compare to test the condition codes.

Unlike most two operand instructions, both operands are required for a comparison, and no result (other than than the setting of the status flags) is produced. The contents of all registers are destroyed.

Note that, unlike the 6502 compare instruction, this comparison is a signed comparison. For example, the two byte integer $0001 is larger than $FFFF, since the first represents 1 and the second represents -1. Signed compares are longer in terms of both space and speed than unsigned compares. If you can avoid using a signed compare, and can instead use an unsigned compare, refer to the CMPW macro.

Coding Examples:

```
     CMP4  NUM1,*       compares NUM1 to the four byte
!                        integer on the top of the
!                        evaluation stack; the four byte
!                        integer on the stack is removed
!                        from the stack
     CMP2  {P1},#4      compares the two byte integer
!                        pointed to by P1 to 4
```

DIVx Integer Division

Forms:

```
LAB     DIV2    NUM1,NUM2,NUM3
LAB     DIV4    NUM1,NUM2,NUM3
LAB     DIV8    NUM1,NUM2,NUM3
```

Operands:

LAB - Label.
NUM1 - The first argument (numerator) for the division.
NUM2 - The second argument (denominator) for the division.
NUM3 - The result.

Description:

A signed integer division is performed on the two arguments, dividing NUM1 by NUM2. NUM3 is optional, if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of all registers are lost. If NUM2 is zero, the divide by zero error flag is raised. See the ERROR macro for a description of the detection and reporting of such errors.

It is important to realize that this is an integer division, and that the result is an integer; thus, 3/2 is 1, not 1.5.

Coding Examples:

```
     DIV4  #7,NUM1,NUM2  divides 7 by NUM1, placing the
!                         result in NUM2; the third
!                         operand is required, since the
!                         default operand uses immediate
!                         addressing, which cannot be
!                         used by a result
     DIV2  #7,#2,NUM1    an inefficient way of setting
!                         the number NUM1 to 3
```

MODx Integer Modulo Function

Forms:

```
LAB     MOD2   NUM1,NUM2,NUM3
LAB     MOD4   NUM1,NUM2,NUM3
LAB     MOD8   NUM1,NUM2,NUM3
```

Operands:

LAB - Label.
NUM1 - The first argument (numerator) for the operation.
NUM2 - The second argument (denominator) for the operation.
NUM3 - The result.

Description:

A signed integer division is performed on the two arguments, dividing NUM1 by NUM2; the result reported is the unsigned integer remainder. NUM3 is optional; if is is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of all registers are lost. If NUM2 is zero, the divide by zero error flag is raised. See The ERROR macro for a description of the detection and reporting of such errors.

Coding Examples:

```
     MOD2  #7,#2,NUM1    places 1, the remainder from the
!                          division, into NUM1
```

MULx Integer Multiplication

Forms:

```
LAB     MUL2   NUM1,NUM2,NUM3
LAB     MUL4   NUM1,NUM2,NUM3
LAB     MUL8   NUM1,NUM2,NUM3
```

Operands:

LAB - Label.
NUM1 - The first argument for the multiplication.
NUM2 - The second argument for the multiplication.
NUM3 - The result.

Description:

A signed integer multiplication is performed on the two arguments, NUM1 and NUM2. NUM3 is optional; if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of all registers are lost. If an overflow of the signed result occurs, the overflow error flag is raised. See the ERROR macro description for error handling details.

Coding Examples:

```
     MUL4  {P1},#4       multiplies the four byte integer
!                         pointed to by P1 by 4
     MUL2  NUM1,NUM2,NUM3 multiplies the two byte
!                         integers at NUM1 and NUM2,
!                         saving the result at NUM3
```

RANx Integer Random Number Generator

Forms:

```
LAB    RAN2   NUM1
LAB    RAN4   NUM1
LAB    RAN8   NUM1
```

Operands:

LAB - Label.
NUM1 - The result.

Description:

A signed integer is generated by a pseudo-random number generator, and the result saved at NUM1. Since no argument is required, this macro becomes the only integer math macro with a single operand. All registers are destroyed.

The random numbers generated are evenly distributed across the entire range for the size of the integer being generated; for example, two byte random numbers range from -32768 to 32767.

The random number generator should be initialized by using the SEED macro before the first random number macro is generated. The SEED macro is described in the last section.

Coding Examples:

```
     RAN8  NUMBER          places an eight byte random
!                           number at NUMBER
```

SIGNx Integer Sign Function

Forms:

```
LAB     SIGN2   NUM1,NUM2
LAB     SIGN4   NUM1,NUM2
LAB     SIGN8   NUM1,NUM2
```

Operands:

LAB - Label.
NUM1 - The argument.
NUM2 - The result.

Description:

The result is 0 if the argument was zero, 1 if it was positive, and -1 if it was negative. The result is placed at NUM2 if it is coded, and at NUM1 if it is not. No errors are possible. The contents of all the registers are lost.

Coding Examples:

```
     SIGN8 NUM1          replaces NUM1 with the result
     SIGN4 *,NUM2        places the sign of the number
!                         on the top of the stack into
!                         NUM2
```

SQRTx Integer Square Root

Forms:

```
LAB     SQRT2  NUM1,NUM2
LAB     SQRT4  NUM1,NUM2
LAB     SQRT8  NUM1,NUM2
```

Operands:

LAB - Label.
NUM1 - The argument.
NUM2 - The result.

Description:

The result is the integer square root of the argument. NUM2 is optional; if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of all registers are lost. If the argument is 0, so is the result. If the argument is negative, the result is the correct square root for the absolute value of the argument, and the operation exception error flag is raised. See the ERROR macro for information on detection and reporting of the error.

Coding Examples:

```
     SQRT2 #450,NUM2     places 21 in NUM2
     SQRT4 {P1}          replaces the four byte integer
!                         pointed to by P1 with its
!                         square root
```

SUBx Integer Subtraction

Forms:

```
LAB     SUB2    NUM1,NUM2,NUM3
LAB     SUB4    NUM1,NUM2,NUM3
LAB     SUB8    NUM1,NUM2,NUM3
```

Operands:

LAB - Label.
NUM1 - The first argument for the subtraction.
NUM2 - The second argument for the subtraction.
NUM3 - Result.

Description:

The second argument (NUM2) is subtracted from the first argument (NUM1). NUM3 is optional; if it is coded, the result is placed there, if it is not coded, the result is placed at NUM1. The contents of the accumulator are lost. If any operand uses indirect or stack addressing, the contents of the Y register are also lost. If the operation is on four or eight byte integers, the X and Y registers are both lost. For two byte operations, V is set if there is an overflow and cleared otherwise. If the operation is on four or eight byte integers, and an overflow of the signed result occurs, the overflow error flag is raised. See the ERROR macro for a description of how to detect and report this error.

Coding Examples:

```
     SUB2  *,#4           the two byte signed integer on
!                          the top of the stack is
!                          decremented by four
     SUB4  {P1},NUM2,{P2} subtracts the four byte integer
!                          at NUM2 from the four byte
!                          integer pointed at by P1,
!                          placing the result at the
!                          location pointed at by P2
```

Chapter 21: Input and Output Macros

The macros in this section provide for the input and output of the basic data types. All output is sent through the output hook at $36, and all input is received through the input hook at $38. All of the macros described in this section can be found in M6502.I.O. Macros in this section include:

ALTCH	use alternate character set
BELL	beep the bell
CLEOL	clear to end of line
CLEOS	clear to end of screen
COUT	character output
GETx N1,CR	variable input
GET_LANG LANG	get current language
GET_LINFO DCB	get language information
GOTOXY X,Y	position cursor on screen
HOME	form feed
KEYPRESS	read keypress
NAMEADDR ADR	fetch address of command table
NORMCH	select normal character set
PRBL AD1	print blanks
PUTx N1,F1,CR	variable output
PUTCR	carriage return
RDKEY NOCURSOR	read keyboard
READXY	read cursor position
SET_LANG LANG	set language
SET_LINFO DCB	set language information
SIZE	find screen size

ALTCH Select Alternate Character Set

Forms:

LAB ALTCH

Operands:

LAB - Label.

Description:

The ASCII control character $0E is sent to the output device. If the device is a CRT, this enables the alternate character set. Some 80 column boards may not have an alternate character set; see your user's manual. On an Apple //e, the alternate character set is the inverse characters. Printers may also use this code; again, see your user's manual for details. The accumulator is returned with a $0E in it.

See the NORMCH macro for a way to reverse the effect.

Coding Examples:

```
ALTCH                (no operands are required)
```

BELL — Beep the Bell

Forms:

LAB BELL

Operands:

LAB - Label.

Description:

The ASCII control character BEL ($07) is sent to the output device. This beeps the speaker if the CRT is in use; most printers will also make an audible sound. The accumulator is returned with a $07 in it.

Coding Example:

```
BELL            (no operand is required)
```

CLEOL Clear to End of Line

Forms:

LAB CLEOL

Operands:

LAB - Label.

Description:

The ASCII control character GS ($1D) is sent to the output device. If the CRT is the output device, the line is cleared from the cursor to the end of the line. This function will work fine with ORCA, as well as most eighty column board drivers. The accumulator is returned with a $1D in it.

Coding Example:

```
     CLEOL               (no operand is required)
```

CLEOS Clear to End of Screen

Forms:

LAB CLEOS

Operands:

LAB - Label.

Description:

The ASCII control character VT ($0B) is sent to the output device. If the CRT is the output device, the screen is cleared from the cursor to the end of the screen. This function will work fine with ORCA, as well as most eighty column board drivers. The accumulator is returned with a $0B in it.

Coding Example:

```
     CLEOS                (no operand is required)
```

COUT Character Output

Forms:

LAB COUT CHAR

Operands:

LAB - Label.
CHAR - Character to write.

Description:

A character is sent to the current output device. CHAR must be a valid operand for a LDA instruction. A similar function is performed by the PUTC macro, but this one is generally more efficient. The accumulator is returned with the character in it.

Coding Examples:

```
COUT  #'.'          write a .
COUT  CH            write the character at CH
```

GETx — Variable Input

Forms:

```
LAB     GET2    N1,CR
LAB     GET4    N1,CR
LAB     GET8    N1,CR
LAB     GETC    N1,CR
LAB     GETS    N1,CR
```

Operands:

LAB - Label.
N1 - Location to place the variable read.
CR - Carriage return flag.

Description:

The get macros are the standard way of reading information from external devices. It receives all information through the $38 hook, which means that the input stream can be redirected to read from disk drives or other input devices by simply placing the address of the character input subroutine for that device at $36.

N1 is used to compute the effective address where the variable read will be stored. It can be specified as an absolute address, an indirect address, or a stack address. The type of variable being read is specified by which macro is used; the GET2 macro reads in a signed two byte integer, while the GETS macro reads a string. The macros, and the types they input, are:

```
GET2  two byte integer
GET4  four byte integer
GET8  eight byte integer
GETC  character
GETS  string
```

The CR parameter is a flag; simply using the CR keyword is enough to signal that the flag is true. This is normally done by coding

```
CR=T
```

in the macro's operand. If CR is true, the input is followed by skipping to the end of the current line. If CR is not true, the next get macro will use the first character that was not used by the original get macro. For Example,

```
GET2  NUM1
GET2  NUM2,CR=T
```

would read NUM1 and NUM2 from the same input line, while

```
GET2  NUM1,CR=T
GET2  NUM2,CR=T
```

would read the numbers from two separate lines.

The contents of all registers are destroyed.

Coding Examples:

```
     GET2  INT          reads a two byte integer from
!                        the keyboard
     GETC  CHAR,CR=T    reads a character from the
!                        keyboard, then skips characters
!                        until a RETURN is typed
```

GET_LANG Get Current Language

Forms:

LAB GET_LANG LANG

Operands:

LAB - Label.
LANG - Place to store language number.

Description:

Places the number of the current default language at LANG, which must be a valid operand for a store instruction. The X and A registers are destroyed. This command is specific to the ORCA operating system.

Coding Examples:

```
GET_LANG LNUM         fetch language number
```

GET_LINFO Get Language Information

Forms:

LAB GET_LINFO DCB

Operands:

LAB - Label.
DCB - Address of control block.

Description:

ORCA communicates with its compilers, linkers, interpreters and editors through this and the SET_LINFO macro. When one of these programs is called, it can read the parameters that are passed to it by doing a GET_LINFO call. The call works like a call to ProDOS internally, with the JSR instruction being followed by the op code and the address of a control block. The fields in the control block are:

```
name         byte  use

COUNT        0     11; number of fields
SFILE        1,2   address of source file name
DFILE        3,4   address of output file name
PARMS        5,6   address of parameter list
MERR         7     max error level allowed
MERRF        8     max error level found
LOPS         9     operations flags
KFLAG        10    keep flag
OUTF         11    list output flag
SYMF         12    list symbols flag
ERRF         13    list errors flag
ORG          14-17 origin
```

In actual use, SFILE is the name of the input file, and DFILE is the name of the output file. DFILE can be omitted, in which case it has a length of zero. Both of these file names are valid ProDOS file names, and can be partial or full path names. They are stored in the operating system, and are subject to

being wiped out by input or output operations, so they must be read immediately and copied into an internal work space.

The PARMS field is only used when a call is being made to an assembler or compiler. It contains the list of names from the NAMES= parameter list, and starts with a zero if there were none. Like SFILE and DFILE, this field is volatile, and should be copied into a local work area immediately.

MERR and MERRF are used to control when the system will stop in an assemble, link, execute sequence. If control is returned to the system with MERR negative, it treats the situation as a terminal error. The ProDOS mark to get to the line that caused the terminal error should be stored in the ORG field, and the file name should be pointed to by SFILE. If MERR is positive, but greater than MERRF, the system will not continue on with a link or program execution.

LOPS is used by the operating system to keep track of what must be done. Currently, the last three bits are used. Bit 0 is a flag indicating that a compile must be performed, bit 1 indicates a link edit is needed, and bit 2 indicates that the finished program is to be executed. A link editor would always find bit 0 cleared, and should clear bit 1 before returning. An interpreter should clear all bits before returning, since linking is not needed and the interpreter itself did the execution. A compiler or assembler should clear the low order bit before returning if the compile is complete, and leave it alone if it is switching languages.

All languages under ORCA must have an equivalent to the APPEND directive in the assembler. Whenever a new file is encountered, the APPEND handler should check the AUX field of the file header to insure that it matches the value of the fourth byte of the compiler itself, which is the language number for the compiler. If it does, then the compiler should continue. Otherwise, it should close all files, set KFLAG to 3, place the address of the file name in SFILE, and return with bit 0 of the LOPS field cleared. The operating system handles the remaining functions of switching languages.

The KFLAG field is used to indicate what should be done with the output from a compiler or assembler. A value of 0 indicates that the output is not to be saved, while 1 indicates that it is to be saved to an object module whose base name is the ProDOS name pointed to by DFILE. In the later case, the compiler should create a file with a .ROOT suffix containing the first segment to execute, and place all subsequent segments in a file with a .A suffix. A value of 2 in this field means that another language has already

created a .ROOT file, while a value of 3 indicates that at least one alphabetic suffix has been used. If KFLAG is 3, the compiler must search the directory for the highest alphabetical suffix which has been used, and use the next one.

OUTF, SYMF, and ERRF indicate if the compiler, assembler or linker should send the source listing, symbol tables, and error messages to the output device. Zero means suppress the output, one means to send it, and two means that the language should choose its own default.

ORG is the start location for the binary file, and is used only by the linker. On entry to an editor, it is a displacement to move into the file being edited.

The accumulator is destroyed.

Coding Examples:

```
          GET_LINFO COUNT     get language linfo
          RTS

COUNT     DC    I1'11'        parameter count
SFILE     DS    2             addr of source file name
DFILE     DS    2             addr of output file name
PARMS     DS    2             addr of parameter list
MERR      DS    1             max error level allowed
MERRF     DS    1             max error level found
LOPS      DS    1             operations flags
KFLAG     DS    1             keep flag
OUTF      DS    1             list output flag
SYMF      DS    1             list symbols flag
ERRF      DS    1             list errors flag
ORG       DS    4               origin
```

GOTOXY Position Cursor On Screen

Forms:

LAB GOTOXY X,Y

Operands:

LAB - Label.
X - Column number, counting from 0.
Y - Row number, counting from 0.

Description:

The CRT cursor is moved to the indicated location. If Y is larger the 23, it is set to 23; if X is larger than the number of columns on the CRT minus 1 (39 for a 40 column screen, 79 for an 80 column screen) then the cursor is placed at the far right of the screen. Operands can be immediate or absolute. All registers are destroyed.

Coding Examples:

```
     GOTOXY #4,#6          places the cursor on row 6,
!                           column 4
     GOTOXY N1,N2          places the cursor on row N2,
!                           column N1
```

HOME — Form Feed

Forms:

LAB HOME

Operands:

LAB - Label.

Description:

The ASCII control character $0C (FF) is sent to the output device. If the device is a CRT, the screen is cleared and the cursor is placed at the upper left corner of the screen. If the device is a printer, most printers will skip to the top of a new page. (If yours does not, it will be necessary to replace the printer driver in the operating system with one that performs this function through software.) The accumulator returns with a value of $0C.

Coding Examples:

```
     HOME                (no operand is required)
```

KEYPRESS — Read Keypress

Forms:

LAB KEYPRESS

Operands:

LAB - Label.

Description:

Upon return, the accumulator contains a one if a key has been pressed, and a zero if not. This is an ORCA specific call.

Coding Examples:

```
KEYPRESS
CMP   #0
BEQ   NOTPRESSED
```

NAMEADR Fetch Address of Command Table

Forms:

LAB NAMEADR ADR

Operands:

LAB - Label.
ADR - Address of the table.

Description:

Places the address of the command table at ADR. Each entry in the command table starts with a type byte which is zero for a utility, positive for a built in command, and negative for a language. The value is a displacement into an internal table in the command processor for built in commands. For languages, anding the value with $7F will return the language number. The second field is the name of the command, stored as a length count followed by the name in capitol letters. The end of the table is marked by two $00 bytes. The accumulator is destroyed. This is an ORCA specific call.

Coding Examples:

```
     NAMEADR ADR       fetch the address of the table
     RTS

ADR  DS    2
```

NORMCH Select Normal Character Set

Forms:

LAB NORMCH

Operands:

LAB - Label.

Description:

The ASCII control character $0F is sent to the output device. If the device is a CRT, this enables the normal character set, reversing the effect of the ALTCH macro. The accumulator returns with a value of $0F.

Coding Examples:

```
NORMCH              (no operand is required)
```

PRBL Print Blanks

Forms:

LAB PRBL AD1

Operands:

LAB - Label.
AD1 - Number of blanks to print.

Description:

AD1 is the number of blanks to print. It must be a valid operand for a load instruction. If zero, 256 blanks are printed. The A and X registers are destroyed.

Coding Examples:

```
PRBL  #10             print 10 blanks
```

PUTx Variable Output

Forms:

```
LAB     PUT2    N1,F1,CR
LAB     PUT4    N1,F1,CR
LAB     PUT8    N1,F1,CR
LAB     PUTB    N1,F1,CR
LAB     PUTC    N1,F1,CR
LAB     PUTS    N1,F1,CR
```

Operands:

LAB - Label.
N1 - Location to place the variable read.
F1 - Field with.
CR - Carriage return flag.

Description:

The CR parameter has the same meaning and is used the same as for the GETx macro, with the exception that the variable is written instead of read.

N1 still specifies the variable, this time for output. The only change is that immediate addressing is allowed in addition to absolute, indirect and stack. It is also possible to output boolean values via the PUTB macro; this writes the string "true" if the boolean byte is non-zero, and "false" if it is zero.

F1 specifies the field width, which defaults to 0. This specifies the width, in characters, of the field to be written to. If the number of characters generated by the put macro is greater than or equal to the field width, the characters generated are printed as is. If the number of characters are less than the field width, blanks are written to right justify the characters in the field. For example,

```
        PUTC    #'c',#1
```

would simply print a "c" on the screen, while

```
        PUTC    #'c','#3
```

would print two blanks, followed by a "c".

Coding Examples:

```
     PUT2  INT,CR=T       writes the two byte integer to
!                            the CRT with a CR
     PUTSC #'They''''re here...',#20 prints "They're
!                            here..." to the CRT, right
!                            justified in a 20 character
!                            field
```

PUTCR — Carriage Return

Forms:

LAB PUTCR

Operands:

LAB - Label.

Description:

The ASCII control character $0D (CR) is sent to the output device. If the device is a CRT, the cursor is placed at the start of the next line, scrolling the screen to get a new line if that is necessary.

The accumulator returns with a value of $0D.

Coding Examples:

```
PUTCR                (no operand is required)
```

RDKEY — Read Keyboard

Forms:

LAB RDKEY NOCURSOR

Operands:

LAB - Label.
NOCURSOR - Cursor flag.

Description:

The keyboard is read, and the value read returned in the A register. If anything is coded in the NOCURSOR field (immediately after the macro), a cursor does not appear on the screen; otherwise, a cursor is used. This is an ORCA specific call.

Coding Examples:

```
RDKEY                          read with cursor
RDKEY    NO                    read without cursor
```

READXY Read Cursor Position

Forms:

LAB READXY

Operands:

LAB - Label.

Description:

The horizontal position of the cursor is returned in X, and the vertical position (counting from top to bottom) in Y. The upper left corner of the screen is 0,0. This is an ORCA specific call. The accumulator is destroyed.

Coding Examples:

```
READXY
```

SET_LANG Set Language

Forms:

```
LAB     SET_LANG LANG
```

Operands:

LAB - Label.
LANG - Language number.

Description:

LANG must be a valid operand for a load instruction. The accumulator is returned with a 0, and the X register is destroyed. The default language for the ORCA system is set to the specified number. This is an ORCA specific call.

Coding Examples:

```
     SET_LANG #3          set the default language to 3
```

SET_LINFO Set Language Information

Forms:

LAB SET_LINFO DCB

Operands:

LAB - Label.
DCB - Address of the control block.

Description:

This macro is the other half of the GET_LINFO macro. It is used by assemblers, compilers, linkers and editors to pass information back to the operating system. See GET_LINFO for a complete description of the control block and the used for the control block fields. This is an ORCA specific call. The accumulator is destroyed.

Coding Examples:

```
SET_LINFO DCB          set language information
```

SIZE — Find Screen Size

Forms:

LAB SIZE

Operands:

LAB - Label.

Description:

Returns the width of the screen in X and the height of the screen in Y. This operation is specific to ORCA. The accumulator is destroyed.

Coding Examples:

```
SIZE                get screen size
STX   WIDTH
STY   HEIGHT
```

Chapter 22: ProDOS Macros

The ProDOS Machine Language Interface (MLI) provides a very regular set of subroutine calls which allow most of the common functions of disk interface to be performed. Limited clock support is also provided. The macros in this section are primarily designed to "hide the ugly," freeing you from looking up the op codes associated with the ProDOS calls and helping avoid possibly disastrous results of misplacing the addresses of the control blocks (DCB's) usually associated with ProDOS calls.

Each call to the MLI consists of a JSR to $BF00, followed by a one byte op code and a two byte DCB address. All communication is through the DCB; this is where ProDOS gets the inputs for the call, as will as where the outputs are placed. With the exception of the TIME call, all ProDOS calls require a DCB. Even with the time call, two bytes must be included after the op code. As a result, each of the ProDOS macros (except TIME) require a single parameter, the absolute address of the DCB.

The following table summarizes the macros for using the MLI. Complete descriptions of each function, as well as the formats for the DCBs, can be found in the ProDOS reference manual.

Macro Name	Op Code	ProDos Title
ALL_INT	40	Allocate Interrupt
CLOSE	CC	Close
CREATE	C0	Create
DEAL_INT	41	Deallocate Interrupt
DESTROY	C1	Destroy
FLUSH	CD	Flush
GET_BUF	D3	Get Buffer
GET_EOF	D1	Get End of File
GET_INFO	C4	Get File Information
GET_MARK	CF	Get Mark
GET_PREFIX	C7	Get Prefix
GET_TIME	82	Time
NEW_LINE	C9	New Line
ON_LINE	C5	On Line
OPEN	C8	Open
READ	CA	Read
READ_BLK	80	Read Block

RENAM	C2	Rename
SET_BUF	D3	Set Buffer
SET_EOF	D0	Set End of File
SET_INFO	C3	Set File Information
SET_MARK	CE	Set Mark
SET_PREFIX	C6	Set Prefix
WRITE	CB	Write
WRITE_BLK	81	Write Block

In addition, there are three macros used to allocate and deallocate memory. These macros make use of the memory bit map maintained by ProDOS. For specifics about the memory map itself, see the ProDOS reference manual.

All of the ProDOS macros from the table above, as well as the memory management macros described in the remainder of the section, can be found in M6502.PRODOS.

FINDBUFF Find an Unused Memory Buffer

Forms:

LAB FINDBUFF ADR,LEN

Operands:

LAB - Label.
ADR - Location to place start address of memory.
LEN - Amount of memory to find.

Description:

The ProDOS memory map is searched from end to beginning for a block of memory at least as large as LEN. If such a block is found, its address is returned in ADR and the carry flag is set, otherwise the carry flag is cleared and ADR is unchanged. Note that the memory is not reserved, only located. All registers are destroyed.

Coding Example:

```
FINDBUFF LOC,#512   find two pages of memory
```

RELEASE Release a Memory Buffer

Forms:

LAB RELEASE START,LEN

Operands:

LAB - Label.
START - Start location of the memory buffer.
LEN - Amount of memory to release.

Description:

The memory starting an START and extending for LEN bytes is released in the ProDOS memory map. Memory is released and reserved in pages, but the macro will compute the correct pages to release for any number of bytes. All registers are destroyed.

Coding Examples:

```
     RELEASE #STUFF,#10  release STUFF for a read
     RELEASE BUFF,#1024  release 1K buffer whose addr is
!                           in BUFF
```

RESERVE — Reserve a Memory Buffer

Forms:

LAB RESERVE START,LEN

Operands:

LAB - Label.
START - Start of memory to reserve.
LEN - Amount of memory to reserve.

Description:

The memory starting an START and extending for LEN bytes is reserved in the ProDOS memory map. Memory is released and reserved in pages, but the macro will compute the correct pages to release for any number of bytes. All registers are destroyed.

Coding Examples:

```
     RESERVE #STUFF,#10  reserve STUFF
     RESERVE BUFF,#1024  reserve 1K buffer whose addr is
!                           in BUFF
```

Chapter 23: Graphics Macros

The macros described in this section provide a unified approach to graphic output which is loosely based on turtle graphics as presented in Apple Pascal. With the graphics macros presented here, all of the display screens available on the Apple // family of computers can be addressed. It is even possible to treat the text screens as graphics screens.

The package can also be quickly adapted to work on any other pixel oriented graphics device. This is because the high level functions are implemented in a completely general way, making use of three subroutines (plot a point, read a point, plot a point with color mapping) and four variables describing the device. This makes the task of adding a new graphics device fairly straight forward. The source code for the package is necessary to do this; it is available as a separate product. The other effect of using such a general method is not so pleasant - a few of these routines are significantly slower than they could be. The sheer size of including sixteen versions of each subroutine ruled out writing separate, faster versions for each graphics screen. For most of the commands, this is not noticeable, but the clear screens are a bit slow due to their pixel by pixel approach to doing the job. If the entire screen is to be cleared, consider zeroing the area with a block move, instead.

Unless noted otherwise, all macros in the graphics library destroy all registers.

The macros described in this section can be found in M6502.GR.

BB — Bit Block Definition

Forms:

LAB BB PIXELS

Operands:

LAB - Label.
PIXELS - Bit map of the pixels for one row of the block.

Description:

The operand is a quoted string consisting of a series of blanks and some other character, usually an asterisk. Each blank will be represented as a zero, while any other character will be represented as a 1 in a bit field which is right filled with zeros. This conforms to the format needed for a black and white picture for the DRAWBLOCK macro.

Coding Examples:

```
MAN BB        '          '
    BB        '          '
    BB        '   ***    '
    BB        '   ***    '
    BB        '    **    '
    BB        '  *****   '
    BB        ' * *** *  '
    BB        ' * *** *  '
    BB        ' * *** *  '
    BB        '   ***    '
    BB        '   * *    '
    BB        '   * *    '
    BB        '   * *    '
    BB        '          '
    BB        '          '
```

COLOR — Set Pen Color

Forms:

LAB COLOR C

Operands:

LAB - Label.
C - Pen color.

Description:

When FILLSCREEN, FILLSHAPE, PLOT, GROUT or DRAWTO are used, all points placed on the graphics screen will be in the current color. The current color can be set using this macro.

There are a total of sixteen colors available on some graphics displays, all of which may be selected using the COLOR macro. For those displays that have fewer colors, setting the color to an unavailable choice has no effect. The table shown below lists the colors and which screen they are available on. The operand for the COLOR macro can be immediate, absolute, or a text color descriptor; available descriptors are also shown in the table. Text descriptors must be given in uppercase. All registers are destroyed.

Color	Descriptor	Number	Available on:
Black	BLACK	0	all screens
Magenta	MAGENTA	1	low res, double high res
Dark Blue	DBLUE	2	low res, double high res
Purple	PURPLE	3	all but black and white
Dark Green	GREEN	4	low res, double high res
Grey 1	GREY	5	low res, double high res
Blue	BLUE	6	all but black and white
Light Blue	LBLUE	7	low res, double high res
Brown	BROWN	8	low res, double high res
Orange	ORANGE	9	all but black and white
Grey 2	GREY2	10	low res, double high res
Pink	PINK	11	low res, double high res
Light Green	LGREEN	12	all but black and white
Yellow	YELLOW	13	low res, double high res
Aqua Marine	AQUA	14	low res, double high res
White	WHITE	15	all screens

Coding Examples:

```
COLOR BLUE          addressed by color name
COLOR #12           immediate selection of LGREEN
COLOR MYCOLOR       color number is at MYCOLOR
```

COLORMAP Color Map Enable/Disable

Forms:

LAB COLORMAP ONOFF

Operands:

LAB - Label.
ONOFF - ON or OFF.

Description:

The graphics library can map colors as it draws them. This powerful capability gives the ability to insure that a line drawn across a screen is seen, no matter what the background; to reverse the screen, etc. Performance of the package is about cut in half, due to the fact that a point must be read before being written. See SET_COLOR for a way to set the color map.

Coding Examples:

```
COLORMAP ON          enable color map
COLORMAP OFF         disable color map
```

DRAWBLOCK Draw a Block

Forms:

```
LAB     DRAWBLOCK BLOCK,X,Y
```

Operands:

LAB - Label.
BLOCK - Block to draw.
X - Number of pixels wide.
Y - Number of pixels high.

Description:

A block is a series of pixel definitions, proceeding from top to bottom, left to right. Each row of a block must end on an even byte boundary, although the block width does not need to correspond to an even number of bytes. Each pixel is a single bit for black and white screens, or a four bit pattern for color screens. The four bit patterns are the numeric values for the colors, listed under the color macro. DRAWBLOCK dumps the indicated block to the screen at the current pen position, which starts at the upper left corner of the block. The X and Y operands can be immediate or absolute values. The pen is left in its original position. All registers are destroyed.

Black and white blocks are normally defined using the BB macro; they can also be defined with a binary DC statement. Color blocks are defined with hex DC statements. Each hex digit corresponds to a pixel of one of the sixteen available colors.

Coding Examples:

```
DRAWBLOCK MAN,#10,#15 see BB for the block
```

DRAWTO — Draw a Line

Forms:

LAB DRAWTO X,Y

Operands:

LAB - Label.
X - X coordinate to draw to.
Y - Y coordinate to draw to.

Description:

A line of the current pencolor is drawn from the current pen position of the specified X,Y location. The pen is left in the specified location. X and Y must be in the range 0 to 32767; if they are not, the least significant fifteen bits of the value provided will be used. The operands can be immediate or absolute. All registers are destroyed.

Coding Examples:

```
COLOR WHITE           draw a white box around
MOVETO #0,#0          the low res screen
DRAWTO #0,#47
DRAWTO #39,#47
DRAWTO #39,#0
DRAWTO #0,#0
```

FILLSCREEN Fill Screen With a Color

Forms:

LAB FILLSCREEN

Operands:

LAB - Label.

Description:

The current viewport is filled with the current color.

Coding Examples:

```
COLOR BLACK          clear the screen
FILLSCREEN
```

FILLSHAPE — Fill a Shape

Forms:

LAB FILLSHAPE

Operands:

LAB - Label.

Description:

The point at the current pen position, and all points of the same color that are simply connected to it, are filled with the current color. Simply connected means all points that can be moved to by going up, down, left or right.

Coding Examples:

```
MOVETO #10,#10        colors the shape at 10,10
COLOR MAGENTA          magenta
FILLSHAPE
```

FINDXY Find the Pen Position

Forms:

LAB FINDXY X,Y

Operands:

LAB - Label.
X - Place to put the X position of the pen.
Y - Place to put the Y position of the pen.

Description:

The current value of the pen coordinates is placed at X and Y, both of which must be absolute operands. Allow two bytes of storage for each value. Valid ranges are 0 to 32767 for both values. All registers are destroyed.

Coding Examples:

```
FINDXY PENX,PENY    read the color of the
READXY PENX,PENY      current point
```

GROUT — Graphics Output

Forms:

LAB GROUT

Operands:

LAB - Label.

Description:

The standard output hook is changed so that subsequent text output is written to the current graphics screen. Text is written in full color, using the current pen color. This means that color high res screens will display only 20 columns of text, while the black and white screens still display 40 (standard) or 80 columns (extended). At the start of each character, the pen position specifies the upper-left corner of the 7x9 character field. The pen is left one position to the right of the upper-right corner of the letter, ready to write the next character. Terminal control codes are not interpreted.

The character output routine makes use of a 1152 byte table of character shapes containing nine bytes for each of the 128 characters in the ASCII character set. Each character consists of nine seven bit patterns; the first pattern in memory corresponds to the top line of the character. Within each byte, the seven bits appear left justified. They are placed on the screen in the same order read. A table containing the standard ASCII character set is provided in the subroutine library. To replace it with a different table, simply create a table in a program module called SYSCHAR.

Coding Examples:

```
            GROUT
            PUTS  #'This string is drawn in graphics'
            TEXTOUT
            PUTS  #'...but this one is in text.',CR=T

   SYSCHAR  START                   local character set

            DC    B'00010000'
            DC    B'00010000'
            DC    B'00010000'
            DC    B'00010000'
            DC    B'11111110'
            DC    B'00010000'
            DC    B'00010000'
            DC    B'00010000'
            DC    B'00010000'
            :
            :
            :
            END
```

INITGRAPH — Set Up a Screen

Forms:

LAB INITGRAPH

Operands:

LAB - Label.

Description:

The view port for the current graphics screen is set to its maximum value and the pen is placed at 0,0. The pen color is set to black. These functions are separate from those of WRITETO, which defines the screen to be drawn on, but the two are normally used together to initialize a graphics session. WRITETO must be used first.

Coding Examples:

```
WRITETO HIRES2        set up the second hi
INITGRAPH             res screen
VIEW  HIRES2
PENCOLOR BLACK
FILLSCREEN
```

MOVETO — Move the Pen

Forms:

LAB MOVETO X,Y

Operands:

LAB - Label.
X - X position to move to.
Y - Y position to move to.

Description:

The graphics pen is moved to the specified location on the graphics screen. Nothing is actually drawn on the screen; only the starting position for a character write or DRAWTO is changed. X and Y must be in the range 0 to 32767; if they are not, the fifteen least significant bits of the value given will be used. The value moved to does not actually have to correspond to a physical point on the screen, since points plotted outside the window are ignored. The operands can be immediate or absolute. The A register is destroyed.

Coding Examples:

```
MOVETO #10,#10        move the pen to 10,10
```

PLOT — Plot a Point

Forms:

LAB PLOT X,Y

Operands:

LAB - Label.
X - X coordinate to plot to.
Y - Y coordinate to plot to.

Description:

The pen is moved to X, Y, and a single point of the current pen color is drawn there. X and Y must be in the range 0 to 32767; if they are not, the least significant fifteen bits of the value will be used. All registers are destroyed.

Coding Examples:

```
PLOT  #10,#20
```

READXY — Read a Point

Forms:

LAB READXY X,Y

Operands:

LAB - Label.
X - X position to read from.
Y - Y position to read from.

Description:

The accumulator is loaded with the number corresponding to the color of the point at X, Y. X and Y must be in the range 0 to 32767 or the least significant fifteen bits of the value provided will be used. The X and Y registers are destroyed. The cursor is moved to X, Y.

Coding Examples:

See FINDXY.

SET_COLOR — Define Color Map

Forms:

LAB SET_COLOR C1,C2,C3

Operands:

LAB - Label.
C1 - Plot color.
C2 - Screen color.
C3 - Resulting color.

Description:

This macro is used to set up a color map. Even if the COLORMAP macro has been used to enable color mapping, the system starts out working as if color mapping were not being used. SET_COLOR allows one element of the color map array to be changed. The example below illustrates the idea by setting up a color map that will cause exclusive oring on a black and white screen. After setting the color map up this way, drawing something onto a black and white screen twice would result in the original screen, no matter how complicated the original screen or the shape drawn might be.

Coding Examples:

```
SET_COLOR WHITE,WHITE,BLACK set up XOR for B+W screen
SET_COLOR BLACK,WHITE,WHITE
COLORMAP ON
```

TEXTOUT Reset Standard Output

Forms:

LAB TEXTOUT

Operands:

LAB - Label.

Description:

Resets the standard output hook after a GROUT so that future text output is sent to the standard output device. This macro must not be used unless a GROUT has been used first. It is acceptable to use several TEXTOUT macros in a row, so long as GROUT has been used at least once before hand.

Coding Examples:

See GROUT.

VIEW — Show a Screen

Forms:

LAB VIEW NUM

Operands:

LAB - Label.
NUM - Screen number or name to view.

Description:

This macro changes the screen that is displayed on the CRT. It can show any of the ten graphics screens or three text screens. It does not change which screen output is sent to (for that function see WRITETO), only the screen that is viewed.

The operand must be one of the thirteen text descriptors. Text descriptors must be coded in uppercase characters. The text descriptors and the screen that will be shown when these values are used are shown in the table below.

Number	Descriptor	Screen
#1	TEXT	primary 40 col screen
#2	TEXT2	secondary 40 col screen
#3	TEXT80	Apple //e 80 col screen
#4	LORES	primary low res screen
#5	LORES2	secondary low res screen
#6	DLORES	extended low res screen
#7	HIRES	primary hi res screen
#8	HIRES2	secondary hi res screen
#9	DHIRES	extended hi res screen
#10	CHIRES	primary color hi res screen
#11	CHIRES2	secondary color hi res screen
#12	CDHIRES	extended color hi res screen

Primary screens are the screens from $400 to $7FF and the high res graphics screen from $2000 to $3FFF. The secondary screens are located at $800 to $BFF for text and low resolution graphics, and $4000 to $5FFF for

high resolution graphics. Extended high and low resolution graphics double the number of points in the horizontal direction, or increase the number of colors from six to sixteen in high resolution graphics displayed on color screens. Like 80 column displays, these modes are only available on an Apple //e with 128K of memory; unlike text, you must also have a revision B or later motherboard, and have a special jumper installed on your extended 80 column card. (The Apple //c meets these requirements.)

Coding Examples:

```
VIEW  HIRES           these are equivalent
VIEW  #7
```

VIEWPORT Set Graphics View Port

Forms:

LAB VIEWPORT X1,X2,Y1,Y2

Operands:

LAB - Label.
X1 - Lowest X value to use.
X2 - Highest X value to use.
Y1 - Lowest Y value to use.
Y2 - Highest Y value to use.

Description:

The viewport is an imaginary window on the graphics screen that you cannot draw out of. Any plot command which tries to write outside of the current window will do nothing. A line drawn partially in the window and partially out of the window will only show up for those parts that are in the window.

Each graphics screen has a specific physical size. The window cannot be set outside of that physical size. Each graphics screen pixel is numbered from 0, so that X1 and Y1 must be positive. In addition, X2 must be greater than or equal to X1, and Y2 must be greater than or equal to Y1. The maximum sizes for each window are shown in the table below.

Operands may be immediate or absolute. All registers are destroyed.

Screen	Max X	Max Y
standard low res	39	47
double low res	79	47
standard high res	279	191
double high res	559	191
standard color high res	139	191
double color high res	139	191
standard text (40 col)	39	23
double text (80 col)	79	23

Coding Examples:

```
VIEWPORT #0,RIGHT,#0,TOP
```

WRITETO — Write to a Screen

Forms:

LAB WRITETO NUM

Operands:

LAB - Label.
NUM - Screen number or name to write to.

Description:

This macro changes the screen that is written to by the graphics macros. It does not change the screen that is shown on the CRT; for that function see the VIEW macro.

The operand must be one of twelve text descriptors. Text descriptors must be coded in uppercase characters. The text descriptors, the corresponding immediate value, and the screen that will be shown when these values are used is shown in the table below.

Number	Descriptor	Screen
#1	TEXT	primary 40 col screen
#2	TEXT2	secondary 40 col screen
#3	TEXT80	Apple //e 80 col screen
#4	LORES	primary low res screen
#5	LORES2	secondary low res screen
#6	DLORES	extended low res screen
#7	HIRES	primary hi res screen
#8	HIRES2	secondary hi res screen
#9	DHIRES	extended hi res screen
#10	CHIRES	primary color hi res screen
#11	CHIRES2	secondary color hi res screen
#12	CDHIRES	extended color hi res screen

Primary screens are the text screen from $400 to $7FF and the graphics from $2000 and $3FFF. The secondary screens are located at $800 and $BFF for text and low resolution graphics, and $4000 to $5FFF for high

resolution graphics. Extended high and low resolution graphics double the number of points in the horizontal direction, or increase the number of colors from six to sixteen in high resolution graphics displayed on color screens. Like 80 column displays, these modes are only available on an Apple //e with 128K of memory; unlike text, you must also have a revision B or later motherboard, and have a special jumper installed on your extended 80 column card. Check with you dealer if you are not sure on these points. The difference between the color and black and white versions of the high resolution graphics screens is in the way that the plot a point primitives work. If a black and white screen is selected, standard high resolution graphics has a total of 280 points available on each line, and extended graphics has 560, but only two colors are recognized: black and white. If the color mode has been selected, only 140 points exist on each line, but each point can have six colors in the standard mode and sixteen colors in the extended mode. Note that this only effects the way the plotting routine are used, since the number of points and number of colors displayed is a function of whether or not a color screen is used to show the graphics.

Coding Examples:

```
WRITETO CHIRES        these are equivalent
WRITETO #10
```

Chapter 24: Miscellaneous Macros

This section describes all of the macros that did not fit into one of the previous groups. Included are a host of useful logic macros, a note macro for making sounds with the Apple speaker, support for the 65C02 instructions added by Rockwell, and some macros for dealing with the F8 ROM. The macros appear in the following libraries:

M6502.65C02
M6502.LOGIC
M6502.MSC

ASL2 Two Byte Arithmetic Shift Left

Forms:

LAB ASL2 N1

Operands:

LAB - Label.
N1 - Number to shift.

Description:

The two byte number at N1 is shifted left. The least significant bit is replaced with a zero, and the most significant bit is placed into the carry flag. All registers are returned intact.

Coding Example:

```
ASL2  NUM1
```

BBRx — Branch on Bit Reset

Forms:

```
LAB     BBR0    ZP,REL
LAB     BBR1    ZP,REL
LAB     BBR2    ZP,REL
LAB     BBR3    ZP,REL
LAB     BBR4    ZP,REL
LAB     BBR5    ZP,REL
LAB     BBR6    ZP,REL
LAB     BBR7    ZP,REL
```

Operands:

LAB - Label.
ZP - Zero page location to test.
REL - Branch point.

Description:

This 65C02 instruction was included by Rockwell on their version of the 65C02, but is not a part of the standard 65C02 instruction set. It tests to see if the bit specified as part of the opcode is clear in the indicated zero page location. If so, a relative branch is made. This operation is only available on the Rockwell 65C02.

Coding Example:

```
        BBR2  5,THERE
```

BBSx — Branch on Bit Set

Forms:

```
LAB     BBS0    ZP,REL
LAB     BBS1    ZP,REL
LAB     BBS2    ZP,REL
LAB     BBS3    ZP,REL
LAB     BBS4    ZP,REL
LAB     BBS5    ZP,REL
LAB     BBS6    ZP,REL
LAB     BBS7    ZP,REL
```

Operands:

LAB - Label.
ZP - Zero page location to test.
REL - Branch point.

Description:

This 65C02 instruction was included by Rockwell on their version of the 65C02, but is not a part of the standard 65C02 instruction set. It tests to see if the bit specified as part of the opcode is set in the indicated zero page location. If so, a relative branch is made. This operation is only available on the Rockwell 65C02.

Coding Example:

```
        BBS2  5,THERE
```

BGT Branch on Greater Than

Forms:

LAB BGT BP

Operands:

LAB - Label.
BP - Branch point.

Description:

The 6502 does not include an instruction to branch after a comparison on the condition that the register was strictly greater than the memory location. This macro serves the purpose, working just like the normal relative branch instructions.

Coding Example:

```
        BGT   THERE
```

BLE Branch on Less Than or Equal

Forms:

LAB BLE BP

Operands:

LAB - Label.
BP - Branch point.

Description:

The 6502 does not include an instruction to branch after a comparison on the condition that the register was less that or equal to the memory location. This macro serves the purpose, working just like the normal relative branch instructions.

Coding Example:

```
        BLE   THERE
```

BUTTON Read a Game Paddle Button

Forms:

LAB BUTTON BTN,VAL

Operands:

LAB - Label.
BTN - Button number.
VAL - Location to place value.

Description:

BTN is the number of one of the three game paddle buttons, numbered from 0 to 2; it can be specified using an absolute or immediate address. The value returned is negative if the button is being pressed, and positive if it is not. The condition codes are set appropriately, so VAL can be omitted, and the program can follow the macro with a BMI to test for a button being pressed. If VAL is specified, it should be an absolute address - the result is stored there.

Coding Examples:

```
          BUTTON #0             branch to PRESSED if
          BMI   PRESSED         button 0 is being
!                                pressed
          BUTTON NUM,RES RES    is the result of
!                                examining button NUM
```

CNVxy Convert x to y

Forms:

```
LAB     CNV24  N1,N2
LAB     CNV28  N1,N2
LAB     CNV2S  N1,N2
LAB     CNV42  N1,N2
LAB     CNV48  N1,N2
LAB     CNV4S  N1,N2
LAB     CNV82  N1,N2
LAB     CNV84  N1,N2
LAB     CNV8S  N1,N2
LAB     CNVS2  N1,N2
LAB     CNVS4  N1,N2
LAB     CNVS8  N1,N2
```

Operands:

LAB - Label.
N1 - The argument.
N2 - The result.

Description:

These macros are used to convert from one type to another type. By varying the characters substituted for x and y, all of the macros used for conversion can be formed. Since they all use exactly the same protocol, they are described together here.

Both operands can be specified using absolute, indirect or stack addressing. It is also possible to use immediate addressing on the argument, although this is not recommended (it would be more efficient to simply use a constant in its original form). The argument is converted from the type indicated by x to the type indicated by y, and stored at N2. Overflow and underflow errors are checked for during the conversion process; see the ERROR macro description for more on error handling. The second operand can be omitted if the result is to be placed at the same location used for the source. S is a string type, so it is possible to use these macros to convert a

string to a binary number, or to convert a binary number to a string without printing the result.

Coding Examples:

```
          CNV24 INT1,INT2      converts the two byte
!                               integer at INT1 into
!                               a four byte integer,
!                               saving the result at
!                               INT2
          CNV84 *              converts the eight byte
!                               integer on the top of
!                               the stack into a four
!                               byte integer, saving
!                               the result back onto
!                               the stack
          CNV42 {P1}           converts the number
!                               pointed to by P1 from
!                               a four byte integer
!                               to a two byte integer
          CNV2S N1,STR         convert 2 byte integer
!                               N1 to string, saving
!                               at STR
STR       DSTR  ,10
```

CMPW Compare Word

Forms:

LAB CMPW N1,N2

Operands:

LAB - Label.
N1 - First number to compare.
N2 - Second number to compare.

Description:

Does a two byte unsigned comparison of N1 to N2, setting condition flags the same way the CMP instruction does. Unsigned compares are far faster than the signed compare done by the CMP2 macro. The accumulator is destroyed.

Coding Example:

```
CMPW  NUM1,#0
```

DBcn Decrement and Branch

Forms:

```
LAB     DBEQ   R,BP
LAB     DBNE   R,BP
LAB     DBPL   R,BP
```

Operands:

LAB - Label.
R - Register or memory location to decrement.
BP - Branch point.

Description:

This set of macros implements the end of a loop in assembly language. R can be a register or location in memory. It is decremented, and then a branch to BP is performed if the condition specified in the op code is met.

Coding Example:

```
            DBNE  X,LAB          decrement X, branch if
   !                              not 0
            DBEQ  Y,THERE        decrement Y, branch if
0
            DBPL  NUM,TOP        decrement NUM, branch
   !                              if positive
```

DBcn2 Two Byte Decrement and Branch

Forms:

```
LAB     DBNE2  ADR,BP
LAB     DBPL2  ADR,BP
```

Operands:

LAB - Label.
ADR - Address of two byte number to decrement.
BP - Branch point.

Description:

These macros implement two byte forms of the previous decrement macros. The two byte number whose address is specified by ADR is decremented. If the condition specified in the op code is met, a branch is made to BP. The accumulator is destroyed.

Coding Example:

```
            DBNE2 NUM1,TOP          decrement, branch if
!                                    not 0
            DBPL2 NUM1,TOP          decrement, branch if
!                                    positive
```

DEC2 Two Byte Decrement

Forms:

LAB DEC2 N1

Operands:

LAB - Label.
N1 - Two byte number to decrement.

Description:

N1 is decreased by one. The accumulator is destroyed.

Coding Example:

```
DEC2  NUM1
```

DSTR Define String

Forms:

LAB DSTR ADR,LENGTH

Operands:

LAB - Label.
ADR - Character string.
LENGTH - Maximum length of string.

Description:

A string is a set of characters which have both a current and maximum length. The current length is the number of characters currently in the string; it can range from 0 (a null string) to the maximum length. In memory, a string is represented by two count bytes, followed by the character bytes. The first count byte is the maximum length of the string. It specifies the total number of bytes used for the character bytes, so the amount of memory used by a string is the maximum length plus two. The second byte is the current length. The remaining bytes are the ASCII character codes for the characters currently in the string. Unused bytes, which occur when a string is less that its maximum length, do not have predictable values.

The DSTR macro allows the definition of string constants and variables. At least one of the operands must be specified. It is legal to specify both. LENGTH is the maximum length of the string, while ADR is the initial string which will be stored in the area. Both must be constants. ADR should be enclosed in single quote marks; if a quote mark is needed as part of the string, it should be coded four times. If ADR is not coded, the string is initialized with a current length of 0. If LENGTH is not coded, it is initialized to the length of the string given.

Coding Examples:

```
        DSTR  'They''''re here...' initializes a
   !                              constant
        DSTR  LENGTH=80           a variable string with
   !                              maximum length 80;
   !                              requires 82 bytes of
   !                              storage
        DSTR  'one',20            a string with max
   !                              length of 20 and current
   !                              length of 3
```

DW — Define Word

Forms:

```
LAB     DW      ADR
```

Operands:

LAB - Label.
ADR - String to define.

Description:

The string is placed in memory after a count byte which gives the length of the characters. The operand should be enclosed in single quote marks; if the string contains a single quote mark, code it four times.

Coding Examples:

```
            DW    'Here''''s a sample'
```

ERROR — Flag an Error

Forms:

```
LAB     ERROR ERR
```

Operands:

LAB - Label.
ERR - The error number.

Description:

An error is flagged by the error processor. The error processor will begin by calling an error intercept routine called SYSERIN. If the carry flag is clear after that call, the error handler prints the error message and terminates processing. The subroutine library has a subroutine called SYSERIN which simply clears the error flag, so that if a program takes no action, any error is terminal.

Errors can be intercepted by simply providing a subroutine called SYSERIN in the program. Upon entry, the X register will contain an error number from the table below. If terminal error processing is desired, one can then clear the carry flag and return. If the error is intercepted, set the carry flag and return.

Error Number	Message
1	Evaluation Stack Overflow
5	Invalid Operation
6	Division by Zero
7	Overflow
8	Underflow
9	Inexact

Coding Examples:

```
          ERROR #6                flags a division by
!                                  zero
```

FLASH — Flashing Characters

Forms:

LAB FLASH

Operands:

LAB - Label.

Description:

Sets $32 to $7F, so that output through the standard F8 ROM character output routine will print flashing characters on the Apple 40 column screen. This will only work on the Apple //e if the standard character set is being used; the standard character set does not support lowercase letters. See the related INVERSE and NORMAL macros. The accumulator is destroyed.

Coding Examples:

```
        FLASH           (no operands are required)
```

INC2 Two Byte Increment

Forms:

LAB INC2 N1

Operands:

LAB - Label.
N1 - Number to increment.

Description:

Increase the two byte number N1 by one.

Coding Example:

```
        INC2  NUM1
```

INITSTACK Initialize an Evaluation Stack

Forms:

LAB INITSTACK LEN

Operands:

LAB - Label.
LEN - Number of bytes for stack.

Description:

Many of the ORCA macros provide a stack addressing mode. The stack these refer to is a software stack. This macro gives an easy way to allocate the space for that stack and initialize all of the variables associated with it. LEN is the length of the stack, and should be a multiple of 256 for efficient use of memory. 512 bytes is a good value to choose as a starting point, 1024 bytes provides a very substantial stack. Memory for the stack is requested from the free memory in the ProDOS memory map.

This stack is not associated in any way with the hardware stack used by the 6502.

Coding Example:

```
   INITSTACK #1024                set up a large stack
```

INVERSE — Inverse Characters

Forms:

LAB INVERSE

Operands:

LAB - Label.

Description:

Sets $32 to $3F, so that output through the standard F8 ROM character output routine will print inverse characters on the Apple 40 column screen. If a program is written to work under ORCA HOST, the ALTCH macro provides a more standard way of doing this. See the related macros FLASH and NORMAL. The accumulator is destroyed.

Coding Examples:

```
        INVERSE          (no operands are required)
```

Jcn Conditional Jumps

Forms:

```
LAB     JCC     BP
LAB     JCS     BP
LAB     JEQ     BP
LAB     JGE     BP
LAB     JGT     BP
LAB     JLE     BP
LAB     JLT     BP
LAB     JMI     BP
LAB     JNE     BP
LAB     JPL     BP
```

Operands:

LAB - Label.
BP - Branch point.

Description:

Relative branches have a limited range. For the many cases when a conditional branch must be made that is outside of that range, these macros let the branch around a JMP be hidden in an easily readable form. A conditional jump macro is provided for each of the branch conditions available on the 6502 as an instruction, as well as those provided in this macro library.

A conditional jump can go to any memory location.

Coding Example:

```
            JNE   THERE             conditional jump
```

LA — Load Address

Forms:

LAB LA AD1,AD2,R

Operands:

LAB - Label.
AD1 - Place to load the address.
AD2 - Address to load.
R - Register to use.

Description:

The address specified by the second operand is loaded into the location specified by the first operand. Both must be absolute addresses. R is the register used to do the load; it defaults to A. AD1 can be a multiple operand.

Coding Examples:

```
          LA    NUM1,4          sets the two byte
!                                integer NUM1 to 4
          LA    P1,THERE,X      loads the address of
!                                the label THERE into
!                                the pointer P1 using
!                                the X register
          LA    (P1,P2),THERE   loads both P1 and P2
!                                with the address THERE
```

LM Load Memory

Forms:

LAB LM AD1,AD2,R

Operands:

LAB - Label.
AD1 - Place to load the byte.
AD2 - Byte to load.
R - Register to use.

Description:

The value specified by the second operand is loaded into the location specified by the first operand. AD1 must be an absolute addresses; AD2 can be absolute or immediate. R is the register used to do the load; it defaults to A. AD1 can be a multiple operand.

Coding Examples:

```
        LM    NUM1,#4         sets the one byte integer
!                              NUM1 to 4 using the Y
!                              register
        LM    (F1,F2,F3),#0   sets the three flags
!                              to 0
        LM    F1,F2           sets F1 to the value
!                              of F2
```

LSR2 Two Byte Logical Shift Right

Forms:

LAB LSR2 N1

Operands:

LAB - Label.
N1 - Two byte number to shift.

Description:

The two byte number at N1 is shifted right. The most significant bit becomes a zero, and the least significant bit is shifted into the C flag.

Coding Example:

```
            LSR2  NUM1              divide NUM1 by 2
   !                                 (unsigned)
```

MASL Multiple Arithmetic Shift Left

Forms:

LAB MASL ADR,NUM

Operands:

LAB - Label.
ADR - Value to shift.
NUM - Number of times to shift it.

Description:

The one byte value at ADR is shifted left NUM times. ADR can be the accumulator. NUM must be a number.

Coding Example:

```
MASL  A,3              shift accumulator
MASL  MEM,4            shift memory
```

MLSR Multiple Logical Shift Right

Forms:

LAB MLSR ADR,NUM

Operands:

LAB - Label.
ADR - Value to shift.
NUM - Number of times to shift it.

Description:

The one byte value at ADR is shifted right NUM times. ADR can be the accumulator. NUM must be a number.

Coding Example:

```
MLSR  A,3              shift accumulator
MLSR  MEM,4            shift memory
```

MOVE — Move Memory

Forms:

LAB MOVE AD1,AD2,LEN

Operands:

LAB - Label.
AD1 - Source address.
AD2 - Destination address.
LEN - Number of bytes to move.

Description:

LEN bytes are moved from AD1 to AD2, starting at the end of the move range and proceeding toward the beginning. LEN can range from 0 to 255; if 0, 256 bytes are moved. LEN must be specified with absolute or immediate addressing. AD1 and AD2 can both use absolute, indirect or stack addressing. AD1 can also use immediate addressing, in which case the entire destination range is set to the immediate value specified.

Coding Examples:

```
          MOVE  #0,PAGE,#0    sets 256 bytes to 0,
!                              starting at PAGE
          MOVE  HERE,THERE,Q  moves Q bytes from
!                              HERE to THERE
          MOVE  {P1},*,#8     moves 8 bytes from where
!                              P1 is pointing to the
!                              stack; P1 must be in
!                              zero page
```

MOVEx — Long Memory Moves

Forms:

```
LAB     MOVEB           AD1,AD2,LEN
LAB     MOVEE AD1,AD2,LEN
```

Operands:

LAB - Label.
AD1 - Source address.
AD2 - Destination address.
LEN - Number of bytes to move.

Description:

These memory moves are similar to MOVE, described above, except that they are not limited to moving 256 bytes at a time. MOVEB moves from AD1 to AD2, working from the beginning of the area towards the end. MOVEE moves from AD1 to AD2, starting with the last byte in AD1 and proceeding back to the first. The final characters are mnemonic for "move from the beginning" and "move from the end." All operands can be specified with absolute, indirect or stack addressing.

Coding Examples:

```
        MOVEB SOURCE,DEST,#$500 these are equivalent
        MOVEE SOURCE,DEST,#$500  DEST and SOURCE do
    !                             not overlap
        MOVEB {P1},*,{P2}       moves the number of
    !                             bytes specified by the

    !                             two  byte integer
    !                             pointed to by P2 from
    !                             where P1 points to the
    !                             stack
```

NORMAL — Normal Characters

Forms:

LAB NORMAL

Operands:

LAB - Label.

Description:

Sets $32 to $FF, reversing the effect of the related FLASH and INVERSE macros. The accumulator is destroyed.

Coding Examples:

```
NORMAL          (no operands are required)
```

NOTE — Play a Note

Forms:

LAB NOTE KEY,TIME,VOICE

Operands:

LAB - Label.
KEY - The particular piano key note to play.
TIME - How long to hold the note.
VOICE - Selects the voice to use.

Description:

This macro and its supporting subroutine simulate a slightly out of tune piano with four voices. Before examining the operands, it is important to understand the limitations of the sound system in use. The Apple speaker can only be clicked; its volume cannot be controlled, and the shape of the pulse created by the click cannot be controlled. This causes the four voices to be out of tune with each other, and results in a "cracking" from a raspy sound on the lower notes to a clear bell sound on the high notes. This cracking occurs in all four voices, but not on the same note.

With that caveat, let us examine the capabilities of the macro. KEY specifies the note to be played, and can range from 0 to 88. Zero is used for a pause; it makes no sound. The other numbers correspond roughly to the keys of a piano, numbered from left to right. TIME specifies the amount of time that the note will be held in 1/16ths of a second; it can range from 1 to 64 (1/16 second to 4 seconds). Voice can range from 0 to 3, specifying one of four voices. The first corresponds to the actual notes on the piano fairly well, and is the clearest; it clicks the speaker once every cycle. Voice 1 clicks the speaker at the start of each cycle, and again after 1/8 cycle, producing a raspier sound. Voice 2 sounds the echo after 1/4 of a cycle, giving a simulated harpsichord. The last voice sounds echoes after 1/8 and 1/4 of a cycle, giving a full sound. All operands can be immediate or absolute.

Coding Examples:

```
TEST     START
         LM    VNUM,#3          plays through the
LB1      LM    KNUM,#0          scales of all voices
LB2      NOTE  KNUM,#4,VNUM
         INC   KNUM
         LDA   KNUM
         CMP   #89
         BLT   LB2
         DBPL  VNUM,LB1
         RTS

VNUM     DS    1
KNUM     DS    1
         END
```

PAGEx — Set Display Page

Forms:

LAB PAGE1
LAB PAGE2

Operands:

LAB - Label.

Description:

These macros serve the dual purpose of setting the graphics page which is displayed (or the text page, although the lack of a way to write directly to the second text page prevents their normal use there) and setting which half of the 80 column page is being written to on the Apple //e. Be sure that a program sets other soft switches appropriately for the correct use.

Coding Examples:

```
PAGE1           sets the display to page 1
PAGE2           sets the display to page 2
```

PREAD — Read a Game Paddle

Forms:

```
LAB     PREAD PDL,VAL
```

Operands:

LAB - Label.
PDL - Paddle number.
VAL - Value read.

Description:

The value of one of the four game paddles is read (the paddles are numbered from 0 to 3). The PDL operand must be specified; it can use immediate or absolute addressing. VAL is optional - if specified, it must use absolute addressing. Whether or not it is specified, the value read is in the Y register. The accumulator is also scrambled. VAL will range from 0 to 255, depending on the position of the paddle read.

Due to the way that the Apple game paddles work, it is not a good idea to read two paddle values in quick succession. In general, about 0.05 seconds must be allowed to elapse between readings. Although this may seem like a short time, it is actually about 50,000 machine cycles, or about 17,000 typical assembly language instructions. A clear symptom that enough time is not being allowed to elapse is unpredictable or clearly erroneous paddle readings. If a program is suffering from this problem, a quick fix is to insert a pause of 1/16 second (0.0625 second) by playing a note of duration 1 and note 0, which is simply a pause.

Coding Examples:

```
        PREAD #0,VAL1          read paddles 1 and 2
        NOTE  #0,#1,#0         with an appropriate
        PREAD #1,VAL2          pause
```

RAM Set the RAM Card Switches

Forms:

LAB RAM READ,WRITE,BANK

Operands:

LAB - Label.
READ - Read access flag.
WRITE - Write access flag.
BANK - Bank selection.

Description:

The RAM macro allows mnemonic access to the RAM card switches in 64K or larger Apple]][computers. READ defaults to RAM; it selects whether the program will read from RAM or ROM. WRITE defaults to ON; it enables or disables the ability to write to the RAM card memory. PAGE selects which bank of $D000 memory is switched in; it defaults to 1 and can be set to 1 or 2. Any use of the RAM macro will change all of these settings, so be sure and set all flags properly.

Although it is possible to use positional parameters with this macro, it is recommended that keyword parameters be used to take advantage of the mnemonic symbolic parameter names.

Coding Examples:

```
;
;  this setting allows ROM to be copied to the RAM
;  card area
;
         RAM   READ=ROM,WRITE=ON,BANK=1
```

RESTORE — Restore Registers

Forms:

LAB RESTORE

Operands:

LAB - Label.

Description:

Restores the user registers from the hardware stack in the order X, Y, A. This is the reverse of the order in which they are stacked by the SAVE macro, so RESTORE can be used to recover the registers saved by that macro.

Coding Examples:

```
        RESTORE         (no operands are required)
```

RMBx Reset Memory Bit

Forms:

```
LAB     RMB0   ZP
LAB     RMB1   ZP
LAB     RMB2   ZP
LAB     RMB3   ZP
LAB     RMB4   ZP
LAB     RMB5   ZP
LAB     RMB6   ZP
LAB     RMB7   ZP
```

Operands:

LAB - Label.
ZP - Location to modify.

Description:

The RMB instruction is provided on the Rockwell 65C02, but is not available in any other 65xx CPU. It causes the bit indicated in the op code to be set to zero in the zero page location indicated in the operand.

Coding Example:

```
            RMB7  NUM               clears the sign bit
   !                                 of NUM
```

SAVE — Save Registers

Forms:

LAB SAVE

Operands:

LAB - Label.

Description:

Saves the user registers to the hardware stack in the order A, Y, X. They can be recovered with the RESTORE macro, described above. A is destroyed.

Coding Examples:

```
        SAVE            (no operands are required)
```

SEED Random Number Seed

Forms:

LAB SEED N1

Operands:

LAB - Label.
N1 - The seed.

Description:

The SEED macro is used to initialized the random number generator that is used by all of the random number generation macros. The seed is set only one time, before the first random number is required.

To understand what the inputs to the SEED macro should be, it is necessary to first understand a little about how it is used to generate random numbers. The first point is that the random number generators do not really produce random numbers; they produce a stream of highly uncorelated numbers, but the stream of numbers produced is always the same if the same seed is used. A different seed will produce a completely different set of numbers.

This points out the two common types of inputs used to random number generators. The first is a specified seed which does not change; this means that the program will use exactly the same sequence of random numbers each time it is executed. This is generally done when one is debugging a program, and would like to have some measure of repeatability. The second input type is itself a more or less random number. This can be a number entered from the keyboard when the program is executed, or it can be provided through some other means. Fortunately, the Apple has an excellent source of random number seeds, produced by a tight loop in the keyboard input routine that continuously updates a seed value. This is the preferred initial seed value for producing unpredictable sequences of numbers, and is the default value used by the SEED macro.

To get the default random number seed, code the SEED macro without an operand. If an operand is supplied, it should be a two byte integer. It is actually acceptable to use any number as input if you are not using

immediate addressing. The reason is that the SEED macro is really after a two byte bit pattern, which could easily be taken from the least significant part of an eight byte integer. The operand can be specified using immediate, absolute, indirect or stack addressing.

Coding Examples:

```
          SEED                uses the keyboard seed
          SEED  INT           uses the two bytes at
!                              INT as a seed
          SEED  *             takes a two byte
!                              integer from the
!                              stack as a seed
          SEED  {P1}          uses the two bytes
!                              pointed to by P1
          SEED  #8            uses 8 as a seed;
!                              good for debugging
```

SMBx Set Memory Bit

Forms:

```
LAB     SMB0   ZP
LAB     SMB1   ZP
LAB     SMB2   ZP
LAB     SMB3   ZP
LAB     SMB4   ZP
LAB     SMB5   ZP
LAB     SMB6   ZP
LAB     SMB7   ZP
```

Operands:

LAB - Label.
ZP - Memory location to change.

Description:

This instruction is only available on the Rockwell version of the 65C02. It sets the bit indicated in the op code to one in the zero page memory location indicated by ZP.

Coding Examples:

```
            SMB7  NUM            set the sign bit of NUM
```

SOFTCALL Soft Reference Call

Forms:

LAB SOFTCALL SUB

Operands:

LAB - Label.
SUB - Subroutine to call.

Description:

A JSR instruction is issued to the indicated subroutine, but the address is specified by a soft reference (DC S) type DC statement. This allows a subroutine to call another subroutine from a section of code that may not be executed in a given program, but not ask the link editor to bring in that subroutine unless requested elsewhere. This macro finds its only use in subroutine libraries; examples of its use can be found in the subroutine library listings.

Coding Examples:

```
SOFTCALL SYSEROR
```

Appendix A: Error Messages

Error Levels

For each error that the assembler or linker can recover from, there is an error level which gives an indication as to how bad the error is. The table below lists the error levels and their meaning. Each error description shows the error level in brackets, right after the message. The highest error level found is printed at the end of the assembly or link edit.

Severity	Meaning
2	Warning - things may be ok.
4	Error - an error was made, but the assembler or linker thinks it knows the intent and has corrected the mistake. Check the result carefully!
8	Error - no correction is possible, but the assembler or linker knew how much space to leave. A debugger can be used to fix the problem without reassembly.
16	Error - it was not even possible to tell how much space to leave. Reassembly will be required to fix the problem.

Recoverable Assembler Errors

When the assembler finds an error that it can recover from, it prints the error on the line after the source line that contained the error. Only one error per line is flagged, even if there is more than one error in the line. The error message is actually a brief description of the error. In the sections that follow, each of the possible error messages is listed, in alphabetical order. After the error message is a number; this is the error level. (More on that

later.) In the description following the error message, every possible cause for the error is explained, and ways to correct the problem are outlined.

ACTR Count Exceeded [16]

More than the allowed number of AIF or AGO directives were encountered during a macro expansion. Unless changed by the ACTR directive, only 255 AIF or AGO branches are allowed in a single macro expansion. This is a safeguard to prevent infinite loops during macro expansions. If more than 255 branches are needed, use the ACTR directive inside of the loop to keep the count sufficiently high.

Address Length not Valid [2]

An attempt was made to force the assembler to use an operand length that is not valid for the given instruction. For example, indirect indexed addressing requires a one byte operand, so forcing an absolute address by coding

```
LDA (|2),Y
```

would result in this error.

Addressing Errors [16]

The program counter when pass 1 defined a label was different than the program counter when pass 2 encountered the label. There are three likely reasons for this to happen. The first is if, for some reason, the result of a conditional assembly test was different on the two passes; this is actually caused by one of the remaining errors. The second is if a label is defined using an EQU to be a long or zero page address, then the label is used before the EQU directive is encountered. The last reason is if a label has been defined as zero page or long using a GEQU directive, then redefined as a local label. On the first pass in both of these cases, the assembler assumes a length for the instruction which is then overridden before pass 2 starts.

Duplicate Label [4]

1. Two or more local labels were defined using the same name. The first such label gets flagged as a duplicate label; subsequent definitions are flagged as addressing errors. Any use of the label will result in the first definition being used.

2. Two or more symbolic parameters were defined using the same name. Subsequent definitions are ignored.

Duplicate Ref in MACRO Operand [2]

A parameter in a macro call was assigned a value two or more times. This usually happens when both a keyword and positional parameter set the same symbolic parameter. For the macro

```
MACRO
EXAMPLE  &P1,&P2
MEND
```

The call

```
EXAMPLE A,P1=B
```

would produce this error, since P1 is set to A as a positional parameter, then to B as a keyword parameter.

Error in Expression [8]

Either the expression contains an error, such as mismatched parentheses, or the expression had too many terms for the assembler to handle. There is no fixed limit to the number of terms or level of parentheses in an expression, but generally the assembler will handle as many terms as will fit on a line, and about five or six levels of parentheses. Check for any kind of syntax error in the expression itself.

Invalid Operand [8]

An operand was used on an instruction that does not support the addressing mode.

Label Syntax [16]

1. A symbolic parameter was expected in the label field, but one was not found. Symbolic parameters must begin with the & character, and are followed by an alphabetic character and one or more alphanumeric characters. Directives which require a symbolic parameter in the label field are:

```
SETA
SETB
SETC
AMID
ASEARCH
AINPUT
```

2. A directive that requires a label was used without one. The directives which must have a label in the label field are:

START
DATA

3. The label field of a statement contained a string which does not conform to the standard label syntax. A label must start with an alphabetic character, underscore or tilde, and can be followed by zero or more alphanumeric characters, underscore characters, and tildes. Only the first ten alphanumeric characters are actually significant.

4. A macro model statement had something in the label field, but it was not a symbolic parameter. If anything occupies the label field of the statement immediately following a MACRO directive, it must be a symbolic parameter.

Length Exceeded [4]

1. An expression was used in an operand that requires a zero page result, and the expression was not in the range 0..255. If external labels are used in the expression, and the result will resolve to zero page when the linker resolves the references, force zero page addressing by preceding the expression with a < character, like

```
LDA (<LABEL),Y
```

If the expression is a constant expression, correct it so that it is in the range 0..255.

2. A directive which requires a number in a specific range received a number outside of that range in the directive. See specific directive descriptions for allowed parameter ranges.

MACRO Operand Syntax Error [4]

The operand of the macro model statement contained something other than a sequence of undefined symbolic parameters separated by commas. The macro model statement is the line immediately following a MACRO directive. If it has an operand at all, the operand must consist of a list of symbolic parameters separated by commas, with no imbedded spaces.

Missing Operand [16]

The operation code was one that required an operand, but no operand was found. Make sure that the comment column has not been set to too low a value; see the description of the SETCOM directive. Remember that ORCA requires the A as an operand for the accumulator addressing mode.

Missing Operation [16]

There was no operation code on a line that was not a comment. Make sure the comment column has not been set to too small a value; see the SETCOM directive. Keep in mind that operation codes cannot start in column 1.

Misplaced Statement [16]

1. A statement was used outside of a code segment which must appear inside a code segment. Only the following directives can be used outside of a code segment:

```
AIF       AGO       ORG       GEQU
MERR      SETCOM    EJECT     ERR
GEN       MSB       LIST      SYMBOL
PRINTER   65C02     65816     LONGA
LONGI     IEEE      TRACE     EXPAND
ALIGN     TITLE     RENAME    KEEP
COPY      APPEND    MCOPY     MDROP
MLOAD
```

The way to remember this list is that any directive or instruction that generates code or places information in the object module must appear inside a code segment.

2. A KEEP directive was used after the first START or DATA directive, or two KEEP directives were used for a single assembly. Only one KEEP directive is allowed, and it must come before any code is generated.

3. The RENAME directive, which must appear outside of a program segment, was used inside of a program segment.

4. An ORG with a constant operand was used inside a program segment, or an ORG that was not a displacement off of the location counter was used outside of a program segment, or two ORG's were used before the same code segment. See the description of the ORG directive for details on its use.

5. More than one ALIGN directive was used for the same program segment.

Nest Level Exceeded [8]

Macros were nested more than four levels deep. A macro may use another macro (including itself) provided that the macro used resides in the same macro file as the macro that is using it, and provided the calls are not nested more than four levels deep.

No END [8]

A START or DATA directive was encountered before the previous code segment was ended with an END directive. Each code segment must end with the END directive.

Numeric Error in Operand [8]

1. An overflow or underflow occurred during the conversion of a floating point or double precision number from the string form in the source file to the IEEE representation for the number. Floating point numbers are limited to about 1E-38...1E38, while double precision numbers are limited to about 1E-308...1E308. If this error occurs, the assembler will insert the IEEE format representation for 0 on an underflow, and infinity for an overflow.

2. A decimal number was found in the operand field which was not in the range -2147483648...2147483647. Since all integers are represented as four byte signed numbers, decimal numbers must be in the above range.

3. A binary, octal or hexadecimal constant was found which requires more than 32 bits to represent. All numbers must be represented by no more than four bytes.

Operand Syntax [16]

This error covers a wide range of possible problems in the way an operand is written. Generally, a quick look at the operand field will reveal the problem. If this does not help, read the section of the reference manual that deals with operand formats for the specific instruction or directive in question.

Operand Value Not Allowed [8]

1. An ALIGN directive was used with an operand that was not a power of two.

2. An ALIGN directive was used in a program segment that was either not aligned itself, or was not aligned to a byte value greater than or equal to the ALIGN directive used in the program segment. For example,

```
          ALIGN    4
T         START
          ALIGN    4
          END
```

is acceptable, but

```
          ALIGN    4
T         START
          ALIGN    8
          END
```

will cause an error.

Rel Branch Out of Range [8]

A relative branch has been made to a label that is too far away. For all instructions except BRL, relative branches are limited to a one byte signed displacement from the end of the instruction, giving a range of 129 bytes forward and 126 bytes backward from the beginning of the instruction. For BRL, a two byte displacement is used, giving a range of -32765 to 32770

from the beginning of the instruction. BRL is only available on the 65816. For one remedy, see the conditional jump macros in the macro library.

Sequence Symbol Not Found [4]

An AIF or AGO directive attempted to branch, but could not find the sequence symbol named in the operand field. A sequence symbol serves as the destination for a conditional assembly branch. It consists of a period in column one, followed by the sequence symbol name in column 2. The sequence symbol name follows the same conventions as a label, except that symbolic parameters may not be used.

Set Symbol Type Mismatch [4]

The set symbol type does not match the type of the symbolic parameter being set. Symbolic parameters come in one of three types; A (arithmetic), B (boolean) and C (character). All symbolic parameters defined in the parameter list of a macro call are character type. SETA and ASEARCH directives must have an arithmetic symbolic parameter; SETB directives must have a boolean symbolic parameter; and SETC, AMID and AINPUT directives must have a character symbolic parameter in the label field.

Subscript Exceeded [8]

1. A symbolic parameter subscript was larger than the number of subscripts defined for it. For example,

```
        LDA     &NUM(4)
&NUM(5) SETA    1
```

would cause this error. A subscript of 0 will also cause the error.

Too Many MACRO Libs [2]

An MCOPY or MLOAD directive was encountered, and four macro libraries were already in use. The best solution is to combine all of the macros needed during an assembly into a single file. Not only does this get rid of the problem, it makes assemblies much faster. Another remedy is to use the MDROP directive to get rid of macro libraries that are no longer needed.

Too Many Positional Parameters [4]

The macro call statement used more parameters in the operand than the macro model statement had definitions for. Keep in mind that keyword parameters take up a position. For example, the following macro calls must all be to a macro definition with at least three parameters defined in the macro model statement operand.

```
CALL    L1,L2,L3
CALL    ,,
CALL    L1,,L3
CALL    ,L1=A,L3
```

Undefined Directive in Attribute [8]

The S attribute was requested for an undefined operation code, or for an operation code that does not use ON or OFF as its operand. The S attribute is only defined for these directives:

```
ERR     PRINTER   EXPAND
MSB     65C02     IEEE
GEN     65816     TRACE
LIST    LONGA
SYMBOL  LONGI
```

Unidentified Operation [16]

1. An operation code was encountered which was not a valid instruction or directive, nor was it a defined macro. If you are using 65C02 or 65816 instructions, make sure that they are enabled using the 65C02 and 65816 directives. Make sure MCOPY directives have been used to make all needed macros available at assembly time.

2. The first operation code in a RENAME directive's operand could not be found in the current list of instructions and directives.

3. A MACRO, MEND or MEXIT directive was encountered in a source file.

Undefined Symbolic Parameter [8]

An & character followed by an alphabetic character was found in the source line. The assembler tried to find a symbolic parameter by the given name, and none was defined.

Unresolved Label not Allowed [2]

1. The operand of a directive contains an expression that must be explicitly evaluated to perform the assembly, but a label whose value could not be determined was used in the expression. In most cases, local labels cannot be used in place of a constant. Even though the assembler knows that the local label exists, it does not know the final location that will be assigned by the link editor.

2. The length or type attribute of an undefined symbolic parameter was requested. Only the count attribute is allowed for an undefined symbolic parameter.

Terminal Assembler Errors

Some errors are so bad that the assembler cannot keep going; these are called terminal errors. When the assembler finds a terminal error, it prints the error message and then waits for a key to be pressed. After a key is pressed, control is passed to the system editor, which loads the file that the assembler was working on and places the line that caused the terminal error at the top of the display screen.

File Could not be Opened

A ProDOS error occurred during an attempt to open a source or macro file.

This is generally caused by a bad file of some type, or a file that is missing entirely. Begin by carefully checking the spelling in the offending statement. Make sure that the file can be loaded with the listed file name using the editor. It is important to specify the pathname the same way as it is listed in the assembler command when doing this check. If the error occurs in a strange place where no files are asked for, keep in mind that a macro file is not loaded into memory until a macro is found - in other words, the problem is in one of the MCOPY or MLOAD directives.

Keep File Could Not be Opened

Either there was not enough memory to open the output file or a ProDOS error was encountered during an attempt to open the output file.

Check the file name used in the KEEP directive for errors. This error will occur if the file name of the keep file exceeds ten characters, since the assembler must be able to append ".ROOT" to the keep file name, and ProDOS restricts file names to fifteen characters.

Symbol Table Overflow

The list that follows outlines the uses made of the symbol table. One or more of the uses will have to be reduced to avoid this error.

1. Each macro in the macro file that is currently open requires twelve bytes. Since only one macro file is open at a time, splitting a macro file into shorter files can help. It is not the length of a macro or the macro file that is a problem, but rather the actual number of macros in a file.

2. Each symbol defined using the GEQU directive requires seventeen bytes of symbol table space. This space is not released at the end of each subroutine. The GEQU directive is only needed for specifying fixed zero page or long addresses; using the EQU directive in a data area and issuing a USING directive for the data area in the subroutine will do just as well for other purposes, and the used symbol table space is released as soon as the data area has been assembled.

3. Each local label in a segment requires seventeen bytes of space. This space is released as soon as the segment has been assembled. Using shorter subroutines will reduce the total number of local symbols in each.

4. Symbolic parameters require a variable amount of symbol table space. Reducing the total number or cutting down on the depth of macro calls can help.

5. The AINPUT directive saves the answers typed from the keyboard in the symbol table. These answers are removed when the segment where the AINPUT directive appears has been assembled. Two ways exist to reduce this kind of use: either split the segment so that fewer AINPUT directives are in any one segment, or answer the questions posed by the directive more briefly.

Unable to Write to Object Module

A ProDOs error was encountered while writing to the object module.

This error is generally caused by a full disk, but could also be caused by a disk drive error of some sort.

Recoverable Linker Errors

When the linker finds an error, it prints the name of the segment that contained the error, how far into the segment (in bytes) that the error was, and a text error message. The descriptions below explain the possible causes for each of the errors, as well as possible ways to avoid the error when the cause is not obvious.

Addressing Error [16]

The segment could not be placed at the same location on pass 2 as it was placed on pass 1. This error is almost always accompanied by some other error which caused this one to occur. Correcting the other error will also correct this one.

If this error occurs with no accompanying error in either the assembly or link edit, check for disk errors by doing a full assembly and link edit. If the error persists, report the problem as a bug.

Address is not in Correct Bank [4]

The most significant, truncated bytes of an expression did not evaluate to a value that matched the current location counter. In all cases except for long addressing, the truncated address bytes must match the location counter. Long addressing is not available on the 6502 or 65C02.

Address is not Zero Page [4]

The most significant bytes of the expression result were not zero, but were required to be zero because of the type of statement in which they were used. Generally, this is used for operands which require zero page results (in the range 0..255), which gives the error its name.

Data Area not Found [2]

A USING directive was issued in a segment, and the link editor could not find a DATA segment by the given name. Either insure that the proper libraries are included, or correct the USING directive.

Duplicate Label [4]

A label was defined twice in the same program. One of the definitions must be removed.

Evaluation Stack Overflow [2]

1. Expression syntax errors can occasionally show up as an evaluation stack overflow. Check that error message for some causes.

2. The expression was too complex to evaluate by the link editor. The expression would have to be extremely complex for this to happen; generally, if the assembler will accept an expression, the link editor will have no problem.

Expression Syntax Error [8]

The format of an expression in the object module was incorrect. This error should not occur unless another assembler or link editor error accompanies it. Correcting the other error will cause this one to go away.

If this error occurs with no accompanying error in either the assembly or link edit, check for disk errors by doing a full assembly and link edit. If the error persists, report the problem as a bug.

Linker Version Mismatch [2]

The link editor is not a recent enough version for the segment being linked. Update the version of the linker being used.

MEM Location has been Passed [4]

A MEM directive was encountered which tried to reserve a memory area which the linker had already passed. The definition must be found by the linker before it has placed code at that location. Move the MEM directive to an earlier subroutine, preferably the first.

ORG Location has been Passed [4]

An ORG was encountered for a location that has been passed. Move the segment to an earlier position in the program.

Relative Address out of Range [4]

The destination of a relative address was too far from the current location.

Undefined Op Code [16]

The linker encountered an instruction that it did not understand. This can occur from one of four causes:

1. The linker in use is an old version, and the assembler or compiler needs a newer version. If this happens, a "Linker Version Mismatch" error will also have occurred.

2. An assembly or compile error caused the generation of a bad object module. Remove all assembly or compile errors.

3. The object module file has been physically damaged. Reassemble to a fresh disk.

4. There is a bug in either the assembler, compiler or link editor. Please report the problem for correction.

Unresolved Reference [8]

A label was referenced but not found by the link editor. If the label shows up in the global symbol table from the link edit, make sure the segment has issued a USING directive for the data segment that contains the label. Otherwise, correct the problem by removing the label reference, or defining it as a global label or in a data segment.

Terminal Linker Errors

Some errors are so bad that the linker cannot keep going; these are called terminal errors. When the linker finds a terminal error, it prints the error message, waits for a keypress, and then quits.

Could not read Sublib Directory

A ProDOS error occurs when attempting to read the sublib directory.

This usually results from a bad disk or disk drive. Try accessing the libraries from a different disk drive, or copy the offending disk to a new one.

File could not be Re-opened

A ProDOS error occurred when trying to re-open an object module file after a disk swap.

Check for disk or disk drive errors.

Illegal Sublib Directory

The sublib directory pointed to by the sublib prefix does not exist, or is not a directory.

Use SET to correct the directory name.

Input File not Found

The .ROOT File could not be Found.

When the linker is invoked by using, for example, LINK MYPROG, it expects to find the file MYPROG.ROOT. That file is created by the assembler (or a compiler). This error occurs when the .ROOT file was not found. Check the spelling of the file name in both the KEEP directive of the assembler and in the link editor command line. Make sure that the file is in the prefix that is specified in the link command.

Object Module Read Error

A ProDOS error occurred during an attempt to read from the currently opened object module. This can sometimes occur after a non-terminal error; if so, get all of those errors out before being concerned about this one. Usually, the error occurs as a result of a bad disk or disk drive.

Out of Memory

The linker needed memory, but all of it had been used.

This error should not occur. If it does, report the problem as a bug.

Output Error

A ProDOS error occurred when trying to write to the binary output file.

This is generally caused by a full disk. If that is not the case, check for a bad disk or disk drive.

Output File Could not be Opened

A ProDOS error occurred during an attempt to open the binary output file.

This is generally caused by a full disk or trying to write to a disk that is write protected. A drive error or trying to write to an unformatted disk could also cause the problem.

Output File could not be Re-opened

After swapping disks, the linker was unable to re-open the output file.

Check for a full disk or a disk or disk drive error.

Symbol Table Overflow

The symbol table could not hold all of the symbols needed by the program.

This should be a very rare error. If it occurs, the only solution is to cut down on the number of global labels in the program. Global symbols are created and passed to the link editor by START, DATA, ENTRY and GEQU directives. Labels created inside of data areas are also passed to the linker, and so take up space.

Appendix B: File Formats

Overview

The ORCA system makes use of four kinds of files: ProDOS TXT files, ORCA SRC files, ORCA OBJ files, and ProDOS BIN files.

Text Files

ProDOS TXT files and ORCA SRC files have the same internal format. Both are a sequence of ASCII characters with lines separated by $0D carriage return codes. Although ProDOS makes no strict requirement, the high bit must be off for use with ORCA programs. In addition, ORCA languages on eight bit Apple // computers will ignore any character beyond the 80th character in each line. Both types of file can be created and changed by the system editor, described in Chapter 12. Most other ProDOS based editors will also suffice, although it will probably be necessary of convert the file to a TXT file before editing, and back to a SRC file afterward.

The difference between TXT and SRC files is entirely in the way the AUX field in the file header is used. TXT files leave this field undefined, while ORCA SRC files define it to be the language number. In addition, ORCA SRC files have a file type of $B0, rather than the $04 used by ProDOS.

Object Modules

ORCA languages take source files as input and produce object modules as output. These object modules are then used as input to the link editor. Object modules are contained in a special file type with a file type number of $B1, which shows up as OBJ when cataloged from ORCA.

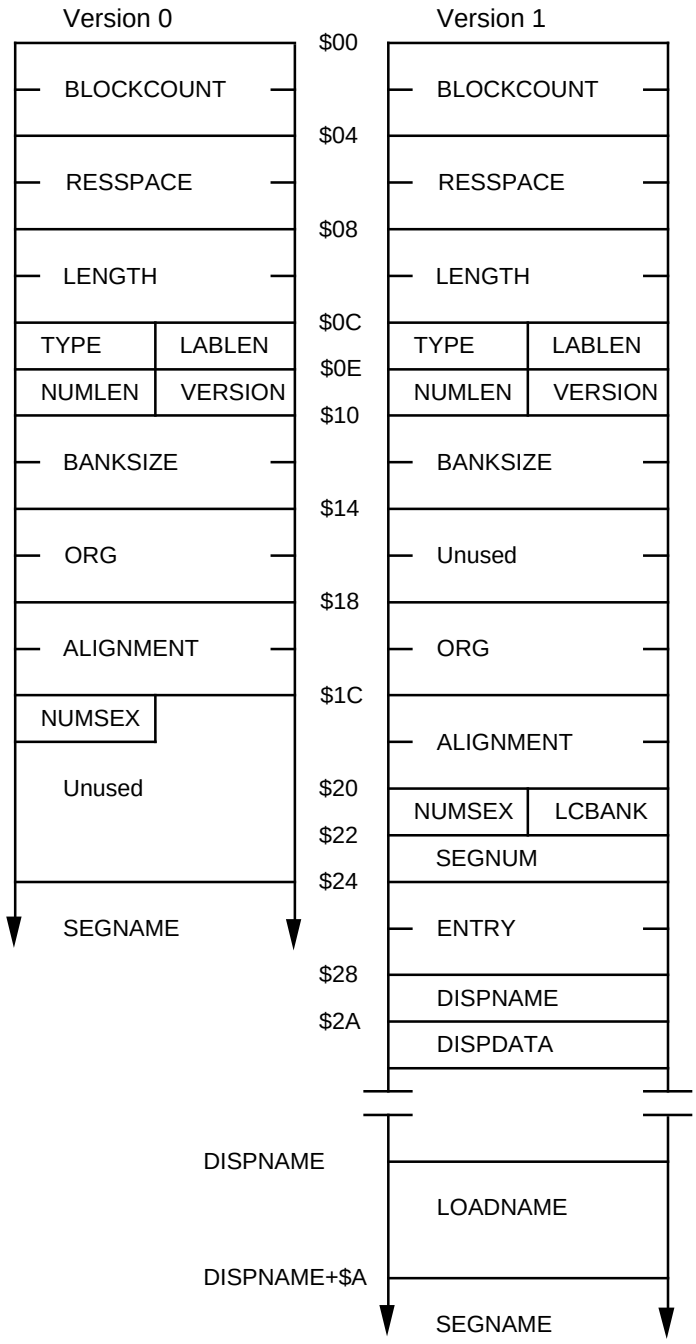
There are now two versions of the object module format. The first, used with ORCA/M 4.0, is labeled as version zero in the header. The second, used by ORCA/M 4.1 on both eight and sixteen bit Apple II computers, and by the Apple IIGS Programmers Workshop on the Apple IIGS, has a one as the version number. Both formats are supported by this version of ORCA/M. The assembler generates object modules using version one of the object module format, while the linker and utilities will take either version zero or version one.

Appendix B: File Formats

The description below describes the subset of version one object modules used by this version of ORCA/M. Several features that deal with relocatable code and libraries have been ommited. For a description of these, see the reference manual for the sixteen bit versions of ORCA/M.

Object modules consist of one or more object segments. Each object segment corresponds to a code or data segment in the assembly language source file. Object segments are aligned to an even block boundary in the object file. The object segment consists of two parts, the header and the body. The header describes the entire object segment. It contains such things as how many blocks the segment occupies, how many bytes it will add to the binary output file, and so on. The second part is one or more operation codes that tell the linker what to put in the final executable file. These operation codes can define constant bytes, give the linker an expression involving external labels to use to compute the value of an address, and so forth.

The headers for the two formats of the object module format are shown below. Note that NUMLEN and LABLEN can greatly affect the size of the op code portion of the object segment.


Version 0
Version 1
$00
BLOCKCOUNT
BLOCKCOUNT
$04
RESSPACE
RESSPACE
$08
LENGTH
LENGTH
$0C
TYPE
LABLEN
TYPE
LABLEN
$0E
NUMLEN
VERSION
NUMLEN
VERSION
$10
BANKSIZE
BANKSIZE
$14
ORG
Unused
$18
ALIGNMENT
ORG
$1C
NUMSEX
ALIGNMENT
Unused
$20
NUMSEX
LCBANK
$22
SEGNUM
$24
SEGNAME
ENTRY
$28
DISPNAME
$2A
DISPDATA
DISPNAME
LOADNAME
DISPNAME+$A
SEGNAME

Name	Description
BLOCKCOUNT	Number of blocks occupied by the segment.
RESSPACE	Number of bytes to place at the end of the segment.
LENGTH	Total number of bytes occupied by the segment in the executable file.
TYPE	Type of segment. In this version of ORCA, only two types are used. $00 indicates a code segment, and is produced by the START directive. $01 indicates a data segment. It is produced by the DATA directive.
LABLEN	Length of labels. This can range from zero to ten. A label length of zero indicates that labels are variable length. Variable length labels start with a length byte and are followed by ASCII characters. They can be up to 255 bytes long. Fixed length labels are padded on the right with spaces. In this version, labels are ten characters long.
NUMLEN	Numbers can be one to four bytes in length. In this version, they are four bytes long.
VERSION	The version of the object module format. Both zero and one are supported. The

	assembler generates object modules using version one.
BANKSIZE	The maximum size of an executable segment. In this version, this value is $10000, for executable segments up to 64K in length.
ORG	Fixed origin for the segment. If omitted, the segment is placed immediately after the one before it. If none have appeared before it, and no origin is specified through another mechanism, the segment will be placed at $2000.
ALIGNMENT	Byte boundary to align the segment to. This value must be a power of two.
NUMSEX	Specifies the order that numbers appear in. If NUMSEX is zero, numbers appear least significant byte first. If it is not zero, they appear most significant byte first. NUMSEX is always zero in this version of ORCA.
LCBANK	Used on the Apple IIGS to indicate if a segment should be loaded into the language card area. Not used in this version of ORCA.
SEGNUM	In the Apple IIGS load segment, this number is used by the loader to check the contents of the file. It is not used in this version of ORCA.

ENTRY	Displacement into the object segment where execution should begin. Not used in this version of ORCA.
DISPNAME	Displacement to the name field. Using a displacement allows the format to be expanded later.
DISPDATA	Displacement to the data field. Using a displacement allows the format to be expanded later.
LOADNAME	Name of the load segment. This field is always ten bytes long. It is not used in this version of ORCA.
SEGNAME	Name of the object segment.

The op code portion of the segment consists of three major groups of op codes. These are:

Op Code	Use
00	end of segment indicator
01-DF	absolute bytes
E0-FF	directives

As indicated, the segment ends when a $00 is found. A hex number from $01 to $DF is essentially a byte counter. The indicated number of bytes are placed directly in the load module, unchanged. The bytes follow the byte count in the order of use.

The last item to describe is the directives themselves.

Op Code	Description
E0	Align - The operand is a NUMLEN byte number indicating the even byte boundary to align to.

Alignment to a page boundary would appear, with M=2, as E0 00 01.

E1 ORG - The operand is a NUMLEN byte number indicating an absolute address to ORG to. If ORG>* (* is the current location counter) zeros are inserted to reach ORG. If ORG=*, no action is taken. If ORG<*, the linker must back track in the binary load module. If ORG < START, where START is the first byte generated, an error results.

E4 USING - followed by a LABLEN byte name of a data area to use.

E5 STRONG - followed by a LABLEN byte name to generate a strong reference to. This does not generate code, but will flag an error if the routine cannot be found. Its effect is to insure that the routine is included in the load module.

E6 GLOBAL - followed by a LABLEN byte label whose value is set to *. The name is followed by three one byte fields. The first two define the length and type attributes of the label. The last is a private flag, which indicates if the name is visible outside of the object module in which it appears (the value is zero) or only in the object file (the value is one).

E7 GEQU - The operand is a LABLEN byte name, followed by three one byte fields, followed by an expression. The first two attributes are the length and type attributes of the label. The last is a private flag, which indicates if the name is visible outside of the object module in which it appears (the value is zero) or only in the object file (the value is one). The value of the label is set to the value of the expression. In this version of ORCA, the expression is always a constant. See EXPR (op code $E8) for a description of an expression.

Note that in version zero of the object module format, the last term is a NUMLEN byte constant.

E8 MEM - The operand is two absolute NUMLEN byte values specifying an absolute range of memory which must be reserved. This is intended for graphics use.

EB EXPR - The first operand byte is the number of bytes to generate, and is <= NUMLEN. This is followed by a reverse polish notation expression. Operators include:

Operator	Description
00	end of expression
01	signed integer add
02	signed integer subtract
03	signed integer multiply
04	signed integer divide
05	modulo operation
06	arithmetic negation
07	signed bit shift operator
08	logical and
09	logical or
0A	logical eor
0B	logical not
0C	logical <=
0D	logical >=
0E	logical <>
0F	logical <
10	logical >
11	logical =

Operands are

80	current location counter
81	ABS - followed by a NUMLEN byte absolute value

82	WEAK - followed by a NUMLEN byte weak reference label name
83	reference to a LABLEN byte label in the operand, resolves as the label value
84	length attribute of the following LABLEN byte label
85	type attribute of the following a NUMLEN byte label
86	count attribute of the following LABLEN byte label
87	REL - followed by a NUMLEN byte displacement from the start of the current module

EC ZPEXPR - same as EB, except that the bits truncated to allow the final value to fit into the specified space must all be zero. This is effectively a variable length implementation of a zero page protocol.

ED BKEXPR - same as EB, except that the bits truncated to allow the final value to fit into the specified space must match the corresponding bits in the correct location counter. This allows checking to insure that an address is in the correct bank.

EE RELEXPR - the first byte is the number of bytes to generate, and is <=NUMLEN. This is followed by a NUMLEN byte displacement from the current location counter, which is the origin for a relative branch. An expression of the same format as that for EB follows this value. The

	expression is resolved as a NUMLEN byte absolute address, then a relative branch is generated from the origin to the computed destination. The result is truncated to the needed number of bytes, and checked to insure that no range errors resulted from the truncation.
EF	LOCAL - same as E6 except that it is a true local label, and is ignored by the link editor unless the module is a data area. In subroutines, it can be used for symbolic debugging.
F0	EQU - same as E7, except that this is a local label, significant only in data areas.
F1	DS - the operand is a NUMLEN byte number indication how many zero bytes of insert in the file at the current location counter.

Binary Files

The output from the link editor is a standard ProDOS binary file. Binary files are a sequence of bytes with a start location given in the file header. When binary files are executed by ProDOS or ORCA, they are loaded unchanged into memory at the given start address, and a JSR is performed to that start address. If a program does not call any special ORCA operating system functions, it can be executed under ProDOS. If it does not interfere with the memory from $800 to $1FFF, it can be executed under ORCA. The program should end with an RTS, which will pass control back to the operating system.

Appendix C: Differences Between ORCA/M 3.5 and ORCA/M 4.0

This appendix gives a quick summary of the differences between ProDOS ORCA (ORCA/M 4.0) and DOS ORCA (ORCA/M 3.5). Most of the changes were actually made when ORCA 3.6 was released - that version of ORCA supports the 65816 under DOS.

Generalities

There are three basic changes which drive all of the minor differences between the two systems. The first is support for the 65816 CPU, including the extension of the address bus from sixteen to thirty-two bits. In the process of adding the RENAME directive, all op code lengths were extended to eight characters. This allowed the op codes for a number of directives to be spelled out, since they were no longer limited to five characters. Finally, some changes were necessary to support ProDOS as opposed to DOS.

Changes to Support ProDOS

All directives that use file names in their operand must now use names that are valid under ProDOS. Unless file names are longer than fifteen characters or unless they contain non-alphanumeric characters, this will probably amount to meaning that the volume parameters will need to be removed. If desired, full or partial path names are now valid.

The directives that require a file name in the operand, and thus need to be checked for possible changes, are:

```
KEEP
COPY
APPEND
MCOPY
MDROP
MLOAD
```

Changes to Support the 65816

The 65816 Directive

A new directive enables the instruction set of the 65816, in much the same way that the 65C02 directive enables the 65C02 instruction set in version 3.5 of ORCA. The directive is called 65816, and its operand is either ON or OFF.

Address Mode Specification

There are four cases on the 65816 which require a syntax different from the operand formats already defined for the 6502. These operand formats comply with the assembler syntax adopted by the Western Design Center as standard for the 65816.

The first new addressing mode is stack relative addressing. The effective address for a stack relative address is computed by adding the sixteen bit stack pointer to an eight bit displacement. Since this is, at the machine level, simply a form of zero page indexed addressing using a somewhat unusual index register, the operand is specified like zero page indexed addresses on the 6502:

```
LDA   DP,S
```

The DP portion of the above example is an expression which must resolve to a one byte number, and the S indicates that the indexing will be off of the S, or stack, register.

Continuing with this line of reasoning, stack relative indexed addressing is a combination of two old addressing modes, indexed indirect addressing and indirect indexed addressing. The addressing mode, again for a LDA instruction, is specified as

```
LDA   (DP,S),Y
```

where DP must again resolve to a one byte number. The effective address for this instruction is computed by adding the sixteen bit stack register to the one byte direct address, whose result points to an address. Y is then added to the address, which becomes the effective address for the instruction.

Two new addressing modes allow a three byte long address to be used for an indirect address. These addressing modes are direct indirect indexed long and direct indirect long addressing. In both of these cases, the fact that a long address is being used is indicated by using square brackets instead of parentheses to indicate the indirection. The following examples illustrate this point.

```
ZP        EQU   4
          LDA   (ZP),Y    direct indirect indexed
!                          (two byte address at ZP)
          LDA   [ZP],Y    direct indirect indexed long
!                          (three byte address at ZP)
          LDA   (ZP)      direct indirect
!                          (two byte address at ZP)
          LDA   [ZP]      direct indirect long
!                          (three byte address at ZP)
```

The last added addressing mode which requires a new operand format is the operand used by the string move instructions MVN and MVP. Both of these operation codes are followed by two one byte values when the machine code is examined. These one byte values represent the bank bytes for the destination and source addresses, respectively. If both the source and destination bank are the current bank, the entire operand becomes optional. If either of the banks is different, the operand is required: it is two absolute long values separated by a comma. The first of these values is in the source bank, while the second is in the destination bank. The examples below show the machine code generated to the left of the instructions, assuming that the code is linked to bank 5.

```
        ABC       EQU   $50000
        DEF       EQU   $60000
540505            MVN
540605            MVN   ABC,DEF
```

Operand Size Specification

The remaining addressing modes added by the 65816 instruction set are all long addressing mode versions of existing instructions, like the absolute long indexed by X, which is the long addressing mode format of the absolute indexed by X addressing mode found on the 6502. These are coded exactly like their absolute addressing mode counterparts. The assembler can automatically select between zero page addresses, absolute

addresses, and absolute long addresses if the address is known at assembly time.

In any case, it is possible to force zero page, absolute or long addressing by using the characters < | or > right in front of the expression in the operand field. (On Apple][+ computers, the ! may be substituted for the | character, which is not on the keyboard.) The < character forces direct page addressing, the | forces absolute addressing, and the > forces long addressing. In cases where the operation code explicitly requires a particular address length, like JML, which requires long addressing, or indirect indexed addressing, which requires a zero page address, forcing an incorrect addressing mode will result in an error. The JML instruction, and others like it, will force long addressing by themselves; it is not necessary to explicitly force long addressing.

```
LDA      <ZP   forces zero page addressing
LDA      |ZP,X forces absolute indexed addressing
LDA      >ZP   forces long addressing
```

Word Size Directives

When the immediate addressing mode is used, the assembler needs a clue as to what mode the 65816 will be operating in when the instruction is executed, so that it will know whether to use a one or two byte immediate value. There are two directives for this purpose, LONGA and LONGI, both of which can have operands of ON or OFF. LONGA specifies the size being used for accumulator and memory based instructions, while LONGI specifies the length of index registers. It is entirely the responsibility of the programmer to insure that the assembler is kept informed of the appropriate register sizes. The following code should clarify the use of these directives.

```
                LONGA ON          tell the assembler
       !                          to use long A
C220            REP   #%00100000  tell the 65816 to
       !                          use long A
                LONGI OFF         tell the assembler
       !                          to use 8 bit X, Y
E210            SEP   #%00010000  tell the 65816 to
       !                          use 8 bit X, Y
A93412          LDA   #$1234
A034            LDY   #$1234
                LONGA OFF         tell the assembler
       !                          to use 8 bit A
E220            SEP   #%00100000  tell the 65816 to
       !                          use 8 bit A
A934            LDA   #$1234
                LONGI ON          tell the assembler
       !                          to use long X, Y
C210            REP   #%00010000  tell the 65816 to
       !                          use long X, Y
A03412          LDY   #$1234
```

In each case, the number of immediate bytes inserted in the code stream was determined by the LONGA and LONGI directives, while the number of bytes the CPU expected was determined by the REP and SEP instructions. Below are two macros which should make the job of setting and clearing these modes easier; the macros set and clear the modes for both the assembler and the CPU.

```
            MACRO
   &LAB     INDEX &L
            AIF   &L='SH',.A
   &LAB     REP   #%00010000
            LONGI ON
            MEXIT
   .A
   &LAB     SEP   $%00010000
            LONGI OFF
            MEND

            MACRO
   &LAB     MEMORY &L
            AIF   &L='SH',.A
   &LAB     REP   #%00100000
            LONGA ON
            MEXIT
   .A
   &LAB     SEP   #%00100000
            LONGA OFF
            MEND
```

Using these macros, the example could be written as

```
            MEMORY   LONG
            INDEX    SHORT
A93412      LDA      #$1234
A034        LDY      #$1234
            MEMORY   SHORT
A934        LDA      #$1234
            INDEX    LONG
A03412      LDY      #$1234
```

Byte Selection Functions

The up arrow character has been added to the less than and greater than keys for use in selecting the byte or bytes to be used during immediate addressing. The function of the various byte selection functions has also been increased slightly to define their operation for two byte immediate values. In the new definition, the > selector causes the immediate value to be divided by 256, placing the most significant byte of a two byte address in the least significant byte position. For one byte immediate addresses, this conforms to the action of the 6502 assembler syntax definition. If a two

byte immediate address is used, this results in the bank byte showing up in the most significant byte position. The ^ character causes the result to be divided by 65536 instead of 256, thus shifting the bank byte into the least significant byte position.

```
            MEMORY  SHORT
            BANK    $12
A956        LDA     #<$3456
A934        LDA     #>$3456
A912        LDA     #^$3456
            MEMORY  LONG
A95634      LDA     #<$3456
A93412      LDA     #>$3456
A91200      LDA     #^$3456
```

Additional Changes to the Assembler

Case Insensitivity

Lowercase letters may now be used anywhere that an uppercase letter is used. The assembler automatically converts to uppercase before using a character. Thus, ABC, abc and AbC are equivalent labels.

Label Formats

There is no longer a limit to the length of labels; however, only the first ten characters of a label are significant. The underscore character can now be used in labels; it too is not significant. For example, THISLABEL and THIS_LABEL are treated as equivalent by the assembler.

Quoted String Syntax

The double quote character may now be used instead of the single quote character for quoting strings. The opening and closing quote types must, however, match. It is not recommended that the double quotes be used by the programmer; they are intended for use by the macro writer so that strings passed to the macro which contain single quote marks within the string can be properly parsed.

Assembler Statistics

The number of source lines processed and the number of lines generated by macro expansions are now printed at the end of an assembly. The number of page faults is also listed. This is the number of times the assembler had to access the macro file during macro expansions.

Eight Byte Integers

A DC format has been added to support integers up to eight bytes in length. It is now possible to specify any integer length, from one to eight, after the I in a DC statement. Expressions and labels may only be used in one to four byte integers.

```
          DC    I1'1'
          DC    I2'1'
          DC    I5'1'
          DC    I8'1'
```

S Attribute

A new attribute is available which lets the source code detect the current status of the assembler flags set by all of the directives which have operands of either ON or OFF. This is provided mostly to support macros which needed to know if the long or short register sizes had been selected. For example, the code below would distinguish between long or short index registers at assembly time, using the conditional assembly capabilities of ORCA to select the proper code.

```
          AIF   S:LONGI,.A
* Insert code here for 8 bit index registers.
          AGO   .B
.A
* Insert code here for 16 bit index registers.
.B
```

Spelling Changes

The maximum number of characters that could be used in an assembler directive was changed from five to eight, allowing several directive names to be spelled correctly. The &SCNT predefined symbolic parameter

spelling has also been changed, in keeping with the ORCA wide convention of starting all system symbols with the characters SYS. (The purpose of this convention is to prevent accidental label conflicts - a programmer should never start a label with the characters SYS, and since the system always does, there will never be a conflict.)

```
OLD             NEW

&SCNT           &SYSCNT
AINPT           AINPUT
APEND           APPEND
ASRCH           ASEARCH
EXPND           EXPAND
PRNT            PRINTER
SYMBL           SYMBOL
```

You can change all of your old files to the new spellings using the editor's global search and replace command, or use the RENAME directive, described below, to avoid problems with all of the changes except &SCNT.

IEEE Directive

The IEEE directive now defaults to ON instead of OFF. In addition, the 3.5 version's double precision floating point format is no longer available; all double precision numbers will be in IEEE format.

MSB Directive

The most significant bit directive now defaults to off instead of on, causing strings and character constants to be generated with the most significant bit off instead of on. To reverse the effect, start your program with MSB ON.

RENAME Directive

The RENAME directive allows the name of any instruction or directive to be changed during an assembly. The reason for this addition is to allow instructions to be renamed to avoid conflicts between the 6502 instruction set and the instruction set of a CPU which is being implemented as a cross assembler via macros from ORCA. To use the directive, place the current operation code, a comma, and the new operation code in the operand field of the RENAME directive. Limitations are that the new operation code must be eight characters or less, contain no blanks, and that the RENAME

directive must appear outside of a subroutine (i.e. before the first START directive or following an END directive and before the succeeding START directive).

Another more immediate use is to provide compatibility with the old directive spellings from ORCA 3.5. For example, starting a program with the following directives would provide almost complete compatibility with the old version - the only changes needed would be to switch the spelling of any occurrence of &SCNT to &SYSCNT, and to eliminate any of the old ten byte double precision floating point DC directives from the program.

```
SETCOM   20
MSB      ON
IEEE     OFF
RENAME   AINPUT,AINPT
RENAME   APPEND,APEND
RENAME   ASEARCH,ASRCH
RENAME   EXPAND,EXPND
RENAME   PRINTER,PRNT
RENAME   SYMBOL,SYMBL
```

SETCOM Directive

The SETCOM directive uses a constant operand in the range 1 to 80 to set the column that defines the start of a comment field to the assembler. The comment column defines the last column where an operand field can start; it currently defaults to 40, and defaulted to 20 in version 3.5 of ORCA.

TRACE Directive

A trace directive has been added; its operand is either ON or OFF, defaulting to OFF. When ON, all lines are listed, even those that the assembler normally does not list. This is provided principally to aid in debugging macros.

Appendix D: Differences Between ORCA/M 4.0 and ORCA/M 4.1

The Command Processor and Utilities

COPY

The copy command will now accept a file name as a destination. For example, if you type

```
COPY MYFILE /THATDISK/THATDIR/NEWFILE
```

and /THATDISK/THATDIR/NEWFILE is not a directory, but /THATDISK/THATDIR is, the file will be copied to NEWFILE in the indicated directory. As was true before, if a directory is specified as the destination, the file retains its original name.

FILETYPE

A new command called FILETYPE has been added. It is used to change the file type of a disk file. Its most common use is to change BIN files to SYS files so that they can be executed by ProDOS during the boot process.

INIT

The INIT utility now supports any disk that allows the ProDOS initialization call to be made, not simply 5-1/4 inch disks. Specifically, this includes the 800K 3-1/2 inch floppy and most hard disks.

MACGEN

The MACGEN utility now saves the file it is building in a temporary file on the work prefix, writing it to the output file name only after all input files are scanned. This means it is safe to use the output file name as an input file, as when modifying an existing macro library.

The utility will also accept all parameters on the command line.

SET

The SET command has been added to allow changing system options from the command line. It replaces the SYSGEN utility, which has been removed.

COMMANDS

The COMMANDS command is no longer a utility that allows the command table to be edited. Instead, it reads the command table from a standard text or source file, which can be changed using any text editor.

The Assembler

OBJ

The OBJ directive was actually added in later versions of ORCA/M 4.0, but was missed by many because it was not in the original documentation. It allows code to be assembled to execute at a specified location, but to be physically located in the normal code stream. Its companion directive, OBJEND, cancels the effect.

ABSADDR

This directive cause absolute addresses to be printed to the left of the assembly listing. Please read the reference manual for cautions on its use.

INSTIME

The number of cycles needed for an instruction are printed in the listing.

Editors Note:

All indexing from the original manual was lost when Microsoft Word 98 refused to load documents created with earlier versions of Microsoft Word. The index you see here is the text from the original manual, but the page numbers will not match this manual exactly. In general, look after the page number you see here--the original manual was 14 pages shorter than this one due to minor differences in page layout.

A

ABSADDR directive 119, 135, 328
ABS2 macro 181, 182
ABS4 macro 181, 182
ABS8 macro 181, 182
absolute addressing mode 125,126
ACTR directive 55, 163
ADD2 macro 181, 183
ADD4 macro 41, 181, 183
ADD8 macro 181, 183
address mode specification 318
addressing modes 126, 127
  absolute indexed 127
  absolute indirect 127
  absolute long 127
  absolute long indexed 127
  direct page 126
  direct page indexed 127
  macros 176
address type DC directive 138
AGO directive 55, 161, 163, 164
AIF directive 161, 164, 165
AINPUT directive 54, 165
ALIGN directive 38, 135
ALL_INT macro 220
ALTCH macro 193, 194, 269
AMID directive 54, 166
ANOP directive 4, 17, 167
APPEND directive 17, 26, 35, 117, 119, 135, 317
Apple II GS 40
ASEARCH directive 54, 167
ASL2 macro 250
ASML command 10, 28, 74, 75, 103
ASMLG command 10, 74, 75
ASM6502 70
ASSEMBLE command 8, 10, 27, 74, 76
assembler directives 17, 18, 133
assembler, listing speed 120
assembler statistics 322
assemblies, partial 28
assembly langugae instructions 17, 18
assembly language statements 17, 18, 19, 23
attribute 56, 161, 323
attributes,
  count 161
  length 161
  setting 162
  type 162

B

BB macro 226
BBRx macro 251
BBSx macro 252
BCC instruction 125
BCS instruction 125
BELL macro 193, 195
BGE instruction 125
BIN files 27
binary files 314

binary number 130
BLE macro 254
BLT instruction 125
boolean expression 172
boolean variable 178
buffers,
 character 91
 commands 99
 string 91
built in command 69
BUTTON macro 255
byte selection functions 321

C

case insensitive 18, 322
CATALOG command 2, 3, 8, 10, 30, 32 70, 74, 76
CHANGE command 10, 31, 74, 76
character 131, 140, 178
character constant 131
character string 173
characters,
 control 178
 non-keyboard 98
CLEOL macro 193, 196
CLEOS macro 193, 197
clock driver 61
CLOSE macro 220
CMP instruction 125
CMP2 macro 181, 184
CMP4 macro 181. 184
CMP8 macro 181, 184
CMPL command 10, 74, 76
CMPLG command 10, 74, 76
CMPW macro 258
CNVxy macro 256
code segment 20, 149
COLOR macro 227
COLORMAP macro 229
command files 73
command line 61
COMMANDS command 10, 63, 67, 74, 76, 328
commands, entering 71
comment 124, 134
comment field 131
COMPILE command 10, 74, 77
compiler, installing 63
COMPRESS command 10, 74, 77
conditional assembly branches 55
conditional assembly directives 133, 150, 163-173
conditional assembly instructions 157
conditional directives 159
.CONSOLE device 72
console driver 61
control characters 92
COPY command 2, 8, 10, 74, 77, 327
COPY directive 35, 117, 119, 136, 317
copy text 14, 95
COUT macro 154, 155, 193, 198
CPA instruction 125
CREATE command 10, 74, 75
CREATE macro 220
CRUNCH utility 10, 29,30, 74, 77
cursor movement commands 92,96

D

data areas 25
DATA directive 17,25, 136, 147
data segments 25, 136
DBEQ macro 259
DBNE macro 259
DBNE2 macro 260
DBPL macro 259
DBPL2 macro 260
DC directive 4, 17, 22, 23, 36, 38, 119, 136, 143, 146, 163
DC directives 135

Address 138
Binary Constant 139
Character String 140
Double Precision Floating Point 141, 143
Floating Point 140, 143
Hexadecimal Constant 139
Integer 137
Reference an Address 138
Soft Reference 139
DCOPY command 10, 74, 78
DEAL_INT macro 220
debug macros 150
DEC2 macro 261
decimal number 130
DELETE command 10, 13, 74, 78
delete line 13, 94, 98
delete text 13, 14, 95
DESTROY macro 220
device number 32

DISABLE command 7, 10, 74, 78
DISASM command 10, 74, 79
disassembler 109
command descriptions 110, 111-114
DIV2 macro 181, 185
DIV4 macro 41, 181, 185
DIV8 macro 181, 185
DRAWBLOCK macro 226, 230
DS directive 17, 141
DSTR macro 262
DW macro 57, 264

E

EDIT command 3, 8, 10, 74, 79, 89
editor,
information window 91
installing 63
macros 100
repeat feature 96
window control 97
EJECT directive 37, 141
ENABLE command 7, 8, 10, 74, 79
END directive 4, 17, 21, 25, 142
ENTRY directive 17, 20, 24
EQU directive 17, 21, 24, 125, 129, 142
ERR directive 36, 119, 142
error,
levels 291
recoverable assembler 291-299
recoverable linker 301-303
terminal 121
terminal assembler 299-300
terminal linker 303-305
ERROR macro 265
ESC key editor commands 96
EXEC files 31, 32, 33, 70, 73
EXPAND directive 36, 37, 119, 142
expressions 128, 134

F

F8 ROM 178, 269
factor 129
file names 71, 75
FILETYPE command 74, 79, 327
FILLSCREEN macro 227, 232
FILLSHAPE macro 227, 233
FINDBUFF macro 61, 221
FINDXY macro 234
FLASH macro 266, 269, 278
floating point numbers 140, 141, 143
FLUSH macro 220
format control directives 35

G

GBLA directive 52, 168
GBLB directive 52, 168
GBLC directive 52, 169
GEN directive 50, 119, 143, 155
GEQU directive 17, 20, 25, 119, 125, 129, 143
GET2 macro 193, 199
GET4 macro 41, 193, 199
GET8 macro 193, 199
GET_BUF macro 220
GETC macro 193, 199
GET_EOF macro 220
GET_INFO macro 220
GETLN subroutine 177
GET_LANG macro 193, 201
GET_LINFO macro 193, 201, 217
GET_MARK macro 220
GET_PREFIX macro 220
GETS macro 193, 199
GET_TIME macro 220
global label 24, 142, 143
GOTO 26
GOTOXY macro 193, 205
graphics 225-247
GROUT macro 227, 235, 242

H

hard reference 138
hardware stack 179
HELP command 3, 10, 62, 70, 74, 80
hexadecimal numbers 131, 139
HOME macro 193, 206

I

IEEE directive 119, 143, 324
immediate addressing mode 126, 320
implied addressing mode 126
INC2 macro 267
INIT command 10, 74, 80, 327
INITGRAPH macro 237
INITSTACK macro 176, 268
INIT. SYSTEM 67
input redirection 32, 72
insert
  character 98
  line 13, 98
  mode 99
  text 13, 94
INSTIME directive 119, 144, 328
integers 130, 137, 177, 178
integers, eight byte 323
INVERSE macro 266, 269, 278

J

Jcn (jump) 270

K

KEEP directive 4, 17, 20, 86, 118, 119, 144, 317
keep parameter
  command processor 75
  link editor 28, 106
keyboard click 84
KEYPRESS macro 193, 207
keyword parameter 44, 45, 158

L

LA macro 271
label 18, 124, 134
label formats 322
label parameter 56

language name 70
languages prefix 83
LANGUAGE, SHOW 85
LCLA directive 51, 169
LCLB directive 170
LCLC directive 51, 52, 170
libraries prefix 84
library files, creating 106
LINK command 8, 10, 74, 80, 105
link editor 103
linking to several locations 27
LIST directive 36, 37, 75, 118, 119, 144
LIST parameter, link editor 106
listings
  printer 122
  screen 121
LM macro 272
local labels 20
LOGIN EXEC file 32
long addressing mode 126
LONGA directive 40, 144
LONGI directive 40, 145
lowercase charaters 19
LSR2 macro 273

M

MACGEN utility 43, 44, 74, 80, 175, 327
macro 18, 41-58, 153-173
  addressing modes 176
  buffer 153
  data types 176
    boolean variable 178
    character 178
    eight byte integers 178
    four byte integers 177
    strings 178
    two byte integers 177
  definition
    statement 154, 158
    writing 153, 154
  directives 163-173
  editor 100
  files 153
  graphics 225-247
  input & output 193-218
  label field 42
  library 44, 145, 146
  math 181-191
  miscellaneous 249-290
  op code field 42
  operand 42
  paramter passing 49
  ProDOS 219-224
MACRO directive 47, 48, 153, 164, 171
MASL macro 274
MCOPY directive 43, 44, 47, 117, 119, 145, 153, 155, 175, 317
MDROP directive 119, 145, 317
memory map 59
memory usage 179
MEND directive 47, 48, 51, 153, 154, 155, 164, 171
MEXIT directive 49, 153, 171
MEM directive 38, 146
MERR directive 39, 119, 146
MLOAD directive 117, 119, 146, 153, 317
MLSR macro 275
MNOTE directive 49, 171, 172
MOD2 macro 186
MOD4 macro 186
MOD8 macro 186
model statements 155
MONITOR 166
monitor commands 9, 10
MOVE macro 276
MOVEB macro 277
MOVEE macro 277
MOVETO macro 238
MSB directive 39, 119, 146, 324
MUL2 macro 181, 187

MUL4 macro 41, 181, 187
MUL8 macro 181, 187

N

names= 29, 75, 104, 118
NAMEADR macro 193, 208
NEW command 3, 74, 81, 89
NEW_LINE macro 220
NORMCH macro 193, 194, 209
NORMAL macro 266, 269, 278
NOTE macro 279

O

OBJ directive 147, 328
OBJ files 27
object module 104, 307
OBJEND directive 147
octal number 130
ON_LINE macro 220
OPEN macro 220
op code 18
operand 124, 125
  field 19, 133
  formats 126, 127
  size specification 319
operation code 124, 134
ORCA/HOST 65, 66
ORCA SRC files 72, 307
ORCA.SYSTEM 59, 65, 67
ORG directive 37, 38, 75, 147, 148
  assembler 119
  link editor 28, 106
output
  link editor 106
  redirection 32, 72

P

PAGE1 macro 281
PAGE2 macro 281
page numbers 150
partial assemblies 28, 75, 118
pathname 134
PEEK command 10, 74, 81
PLOT macro 239
positional parameter 44, 45, 155, 157
PREAD macro 282
PRBL macro 193, 210
PREFIX command 10, 74, 82
.PRINTER device 72
PRINTER directive 37, 119, 148
printer listing 148
PRINTERINIT string 85
PRINTER_SLOT number 84
ProDOS 59, 60, 317
  binary file 314
  BRUN 103
  memory map 268
  MLI 219
  TXT file 70, 72, 307
ProDOS-8 40
ProDOS-16 40
PUT2 macro 193, 211
PUT4 macro 41, 193, 211
PUT8 macro 193, 211
PUTB macro 193, 211
PUTC macro 193, 211
PUTCR macro 41, 193, 213
PUTS macro 41, 193, 211

Q

QUIT command 10, 74, 82
quit editor 95
quoted string syntax 322

R

RAM macro 283
RAN2 macro 181, 188
RAN4 macro 181, 188
RAN8 macro 181, 188
RDKEY macro 193, 214
READ macro 220
READ_BLK macro 220
READXY macro 193, 213
  graphics macro 240
  text macro 215
redirection, input and output 72
relative addressing mode 127
RELEASE macro 61, 222
RENAM macro 220
RENAME command 8, 10, 74, 83
RENAME directive 119, 148, 149, 324
repeat count 12, 96
replace, search and 14, 15
repalcement string 99
RESERVE macro 61, 223
reserving memory 60
RESTORE macro 284, 286
RETURN 5
RMBx macro 285
Rockwell 65C02 285
.ROOT 104
RUN command 8, 74, 83

S

SAVE macro 284, 286
SCAN command 10, 74, 83
search 14, 93, 99
search and replace 14, 94, 99
search string 99
SEED macro 287
segname 134
sequence symbol 55, 161
SET command 10, 67, 74, 83, 328
  CLICK 84
  LANGUAGES 83
  LIBRARIES 84
  PRINTERINIT string 85
  PRINTERSLOT 84
  SYSTEM 83
  UTILITIES 84
  WAIT 84
  WORK 84
SETA directive 52, 53, 172
SETB directive 52, 53, 172
SET_BUF macro 220
SETC directive 53, 172
SET_COLOR macro 229, 241
SETCOM directive 149, 325
SET_EOF macro 220
SET_INFO macro 220
SET_LANG macro 193, 216
SET_LINFO macro 193, 202, 217
SET_MARK macro 220
SET_PREFIX macro 220
set symbol directives 52
SHOW command 10, 31, 32, 72, 74, 85
SIGN2 macro 181, 189
SIGN4 macro 181, 189
SIGN8 macro 181, 189
simple expression 128
SIZE macro 193, 218
SMBx macro 289
SOFTCALL macro 290
soft reference 139, 290
special function keys 90
spelling changes 3.5 -> 4.0 323
SQRT2 macro 181, 190
SQRT4 macro 181, 190
SQRT8 macro 181, 190
START directive 4, 17, 38, 118, 147, 149
status bit 145
string 134, 140, 262
  constants 262
  manipulation 54

search 93, 94, 99
search and replace 94, 99
variables 262
SUB2 macro 181, 191
SUB4 macro 41, 181, 191
SUB8 macro 181, 191
SUBLIB directory 30
subroutine library 30
subscripted parameters,
actual 159
explicitly defined 159

SWITCH command 10, 30, 74, 85
SYMBOL directive 36, 37, 75, 119, 150
SYMBOL parameter, link editor 106
symbol table, link editor 107
symbolic parameter 49, 52, 56, 57, 155, 157, 158, 159, 160, 162, 170
&SYSCNT 57, 324
system configuration 66
system prefix 83

T

tab 99, 100
term 128, 129
terminal control codes 178
text
copying 14
deleting 13
entering 13, 89
files, types 72, 307
inserting 13
moving 14
TEXTOUT macro 242
TIME, show 85
TITLE directive 36, 37, 150
TOS (top of stack) 176
TRACE directive 150, 160, 161, 164, 165, 325
TYPE command 10, 74, 85

U

UNITS, show 85
uppercase letters 19
USING directive 25, 150
utility 9, 62, 63, 66, 69
/UTILITY disk 3, 9, 62
utility prefix 84

V

VIEW macro 243
VIEWPORT macro 245

W

wildcards 7
WRITE macro 220
WRITE_BLK macro 220
WRITETO macro 211, 237, 246
writing macros 47
word size directives 320
work prefix 84

X

XREF command 10, 74, 86

Z

zero page addressing 126
zero page indexed addressing mode 127

numbers

65802 CPU 40
65816 CPU 40, 126, 144, 317, 319
65816 directive 40, 119, 151, 318
65C02 CPU 39, 40, 119, 151
65C02 directive 39, 119, 151